N° d'ordre : 2252

THÈSE

présentée

pour obtenir

LE TITRE DE DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

École doctorale : **Informatique et Télécommunications**

Spécialité : **Informatique**

Par

Mahmoud NASSAR

Analyse/conception par points de vue : le profil VUML

Soutenue le 28 Septembre 2005 devant le jury composé de :

M.	Jean-Pierre GIRAUDIN	Rapporteur et Président du jury
MM.	Bernard COULETTE	Directeur de thèse
	Jean BÉZIVIN	Rapporteur
	Bernard CARRÉ	Examinateur
	Xavier CRÉGUT	Examinateur
	Abdelaziz KRIOUILE	Examinateur

à ma mère,
à mes sœurs,
à mon épouse

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés au sein de l'équipe ISYCOM (Ingénierie des Systèmes COMplexes) du laboratoire GRIMM (Groupe de Recherche en Informatique et Mathématiques du Mirail) de l'Université de Toulouse le Mirail (UTM) en étroite collaboration avec le Laboratoire LGI (Laboratoire Génie Informatique) de l'Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes (ENSIAS) de Rabat.

Je tiens à remercier très vivement mon directeur de thèse Monsieur Bernard COULETTE, Professeur à l'Université de Toulouse le Mirail, pour son soutien, sa disponibilité, sa patience, la collaboration étroite dans laquelle nous avons travaillé et son aide qui m'ont permis de mener à bien ces travaux. Merci également pour ses relectures minutieuses de ce mémoire.

Je tiens également à assurer ma reconnaissance à Monsieur Jean BEZIVIN, Professeur à l'Université de Nantes, et Monsieur Jean-Pierre GIRAUDIN, Professeur à l'IUT2 de Grenoble pour l'intérêt qu'ils ont témoigné à ces travaux et pour avoir accepté d'être rapporteurs de cette thèse. Je les remercie également pour les remarques et conseils prodigués. Merci aussi à Monsieur Jean-Pierre GIRAUDIN de m'avoir fait l'honneur de présider ce jury.

Merci infiniment à Monsieur Bernard CARRÉ, Maître de conférences à l'Université de LILLE, de m'avoir fait l'honneur d'être membre de jury de cette thèse.

J'exprime également ma profonde gratitude à Monsieur Xavier CRÉGUT, Maître de conférences à l'Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), pour les nombreuses discussions techniques que nous avons eues. Je le remercie également pour ses conseils et ses remarques tout au long de ces années. Je le remercie aussi pour avoir bien voulu participer à ce jury.

Je tiens à exprimer ma plus sincère gratitude à Monsieur Abdelaziz KRIOUILE, Professeur à l'ENSIAS de Rabat, pour m'avoir assisté durant cette thèse. Ses conseils m'ont été d'une grande aide. Je le remercie également de m'avoir fait l'honneur de participer au jury de cette thèse.

Merci à Madame Sophie EBERSOLD, Maître de conférences à l'Université de Toulouse le Mirail, pour les nombreuses discussions que nous avons eues sur le thème de cette thèse et pour ses précieux conseils.

Merci à tous les membres de l'équipe GRIMM/ISYCOM pour leur accueil et la bonne ambiance dans laquelle ils m'ont permis de travailler pendant ces années de thèse. Je les remercie aussi pour leur soutien et leurs commentaires enrichissants lors de mes présentations orales.

Je remercie très vivement tous les partenaires du réseau franco-marocain STIC en Génie Logiciel lancé en 2002 pour leurs remarques et suggestions qui m'ont permis d'affiner et de valider ce travail.

J'adresse également mes chaleureux remerciements à tous les membres du Laboratoire Génie Informatique de l'ENSIAS de Rabat. Leurs conseils, leur amitié et leur bonne humeur m'ont beaucoup encouragé durant mes travaux.

Je remercie tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

Enfin, je tiens à remercier tous ceux qui m'ont soutenu, et plus particulièrement ma famille, mon épouse Salima et mes amis.

Résumé

L'objectif de la thèse est de proposer une extension du langage de modélisation unifié (UML) orientée points de vue. Cette extension, appelée VUML (View based Unified Modeling Language) offre un formalisme (extension d'UML) pour modéliser un système logiciel par une approche combinant objets et points de vue. Le principal ajout à UML est celui du concept de *classe multivues*. Une classe multivues est une unité d'abstraction et d'encapsulation qui permet de stocker et restituer l'information en fonction du profil de l'utilisateur. Elle offre des mécanismes de gestion des droits d'accès aux informations, de changement dynamique de points de vue et de gestion de la cohérence entre les vues dépendantes. De plus, VUML propose un modèle de *composant multivues* qui permet de représenter une classe multivues au niveau du diagramme de composants.

Sur le plan sémantique, VUML étend le métamodèle d'UML et introduit un certain nombre de stéréotypes regroupés sous forme d'un profil UML. A l'instar d'UML, la sémantique VUML comprend un aspect statique et un aspect dynamique. La sémantique statique de VUML est définie par le métamodèle, des règles de bonne modélisation (well-formedness rules) exprimées en langage formel OCL (Object Constraint Language) et des descriptions textuelles informelles. La sémantique dynamique quant à elle est décrite d'une manière informelle.

Sur le plan méthodologique, VUML propose un noyau d'une démarche qui permet d'intégrer de façon logique et consistante la notion de point de vue dans le processus de développement dans le contexte de l'approche MDA (Model Driven Architecture).

L'outil support à VUML a été conçu et réalisé en adaptant l'*atelier Objecteering/UML* par la technique des profils. Cet outil permet de mener une modélisation à base de vues, de vérifier la cohérence des modèles élaborés et de générer du code objet (Java,...) en appliquant un patron d'implémentation générique sur un diagramme de classes VUML.

Mots clés : Modélisation, profil UML, vue/point de vue, classe/composant multivues, approche MDA.

Abstract

This thesis proposes a viewpoint oriented extension of the Unified Modelling Language. This extension, called VUML (View based Unified Modelling Language) provides a formalism for modelling software systems through objects and viewpoints. The main UML extension is the concept of *multiviews class* whose goal is to encapsulate and deliver information according to the user profile (viewpoint). VUML allows the dynamic change of viewpoint and offers mechanisms to manage consistency among dependent views. Moreover, VUML proposes a *multiviews component* model. Such a component allows to represent a multiviews class in a component diagram.

On the semantic level, VUML extends the metamodel of UML and introduces a set of stereotypes grouped in a UML profile. As in UML, the VUML semantics includes static and dynamic aspects. The VUML static semantics is defined by the metamodel, the well-formedness rules expressed in the formal language OCL (Object Constraint Language), and informal textual descriptions. The VUML dynamic semantics is described so far in an informal manner.

On the methodological level, VUML proposes a process that makes possible a logical and consistent integration of the viewpoint concept in the development process according to the MDA approach.

A VUML tool has been designed and implemented by adapting the Objecteering/UML tool through the profile technique. This tool allows to carry out a view based modelling, to check the consistency of the elaborated models and to generate object code (Java...) by applying a generic implementation pattern to a VUML class diagram.

Keywords: Modelling, UML profile, view/viewpoint, multiviews class/component, MDA approach.

Table des matières

Table des figures

Introduction générale

Dans la société de l'information qui se développe à grande vitesse notamment à travers l'Internet, l'accès aux informations doit être ouvert au plus grand nombre de citoyens. Pour cela, cet accès doit être différencié de façon à respecter les cultures, les niveaux de sensibilisation, les différences de vitesse d'apprentissage, les droits du citoyen et la protection des informations personnelles. Dans cette perspective, nous pensons que le développement et la maintenance de systèmes d'informations "centrés sur les points de vue des acteurs" jouera un rôle stratégique dans le futur. A l'instar des chercheurs travaillant dans l'ingénierie des exigences, nous préconisons de prendre en compte les besoins des acteurs d'un système d'information (développeurs, utilisateurs finals, exploitants...) le plus tôt possible dans le processus de développement.

Lors du développement d'un système complexe (Le Moigne, 1990), la construction d'un modèle global prenant en compte simultanément tous les besoins des acteurs est impossible. Dans la réalité, soit plusieurs modèles partiels sont développés séparément et coexistent avec les risques d'incohérence associés, soit le modèle global doit être fréquemment remis en cause, et parfois profondément, quand les besoins des utilisateurs évoluent.

L'objectif que nous poursuivons depuis plusieurs années — à l'IRIT tout d'abord, puis actuellement au sein de GRIMM-Isycom et de ses partenaires de l'ENSIAS — est de proposer une méthodologie d'analyse/conception de système centrée sur les points de vue des acteurs interagissant avec le système. Ces concepts de vue/point de vue — que l'on retrouve également sous les termes voisins de rôle, sujet, perspective, aspect — ont été étudiés dans plusieurs domaines de l'informatique : bases de données, représentation des connaissances, analyse et conception, langages de programmation, outils de Génie logiciel, etc. Les travaux menés par notre équipe à partir de 1993 ont donné lieu à la définition du langage VBOOL (Marcaillou et al., 1994), et à la méthode associée VBOOM (Kriouile 1995, Coulette et al. 1996). Parmi les apports principaux de l'approche VBOOM, on peut citer la modélisation décentralisée selon des points de vue multiples, l'introduction du concept de classe flexible pour supporter les vues de granularité fine, le changement dynamique de point de vue, un mécanisme de gestion de la cohérence entre sous-modèles. Par contre, l'expérience a permis de mettre en évidence plusieurs limites de VBOOM ou de sa mise en œuvre : subjectivité dans la définition des vues, rigidité de l'héritage multiple pour prendre en compte l'évolution d'un modèle à bases de vues, complexité d'une programmation avec le langage dédié VBOOL, manque d'outils supports performants, etc.

C'est pour cette raison que nous avons décidé de reconsidérer l'approche développée dans VBOOM en conservant l'objectif d'une modélisation multivues à granularité fine, et en adoptant les principes suivants : pour chaque type d'acteur (utilisateur final ou non) nous associons d'une manière déterministe un point de vue (qui s'applique sur tout ou partie du système) et une vue unique qui correspond à l'application de ce point de vue sur une entité donnée ; l'implantation et la gestion de l'évolution des vues sont réalisées par l'intermédiaire de la délégation (au lieu de l'héritage multiple). Par ailleurs, prenant en compte l'effet "UML" et sa généralisation comme standard "de facto" dans la modélisation de systèmes logiciels, nous avons opté pour la réutilisation des standards de l'OMG

(OMG-site, 2004) et ciblons les langages à objet du marché. La notion de vue proposée dans UML (Unified Modeling Language) (OMG, 2003a) ne correspond pas à nos besoins. En effet, une vue dans UML est un moyen offert au concepteur pour décrire l'architecture d'un système en fonction des phases de développement (cas d'utilisation, logique, composants, déploiement). Cependant, UML n'offre pas de mécanisme à granularité fine pour intégrer les points de vue des acteurs au cœur même de la modélisation, à savoir dans les diagrammes de classes.

Aussi avons-nous décidé de développer une méthodologie d'analyse/conception dont le noyau est un profil UML, appelé VUML (View based Unified Modeling Language), qui supporte la construction de composants de conception multivues. VUML offre la notion de classe multivues (Nassar et al. 2002, Nassar et al. 2003) qui est constituée d'une base et d'un ensemble de vues spécifiques reliées à la base par une relation d'extension. Elle permet de stocker et de restituer l'information selon le profil de l'utilisateur. Elle offre des possibilités de changement dynamique de point de vue et d'expression des dépendances entre vues. La représentation d'une classe multivues sous forme de composant permet de décrire ses vues comme des interfaces "multivues" fournies et/ou requises par le composant. La description d'une architecture en composants multivues réutilisables permet ensuite le déploiement d'une application multivues. Le choix d'un profil UML permet de s'appuyer sur les mécanismes d'extension du formalisme UML, notamment les stéréotypes, sans remettre en cause le métamodèle UML existant, ce qui ouvre la voie au support de VUML par les principaux outils UML du marché.

Même si à l'origine ce travail ne s'inscrivait pas explicitement dans l'approche MDE/MDA (Model Driven Architecture), nous pouvons constater qu'il s'intègre parfaitement dans cette perspective puisque la construction d'un modèle VUML peut être considérée comme une transformation de modèles UML. Nous proposons ainsi un noyau de démarche qui permet d'intégrer la notion de point de vue dans le processus de développement dans le contexte de l'approche MDA. Ceci permet de mener une analyse/conception par points de vue dirigée par les modèles.

Présentation de ce mémoire

Le travail réalisé est présenté dans ce mémoire selon un découpage en cinq chapitres.

Le premier chapitre présente l'état de l'art du domaine. Dans un premier temps, nous abordons la problématique de la modélisation des systèmes complexes. Nous présentons ensuite le projet VBOOM qui constitue le point de départ de notre approche. Puis nous traitons les principales propositions existantes autour de la modélisation par points de vue. La dernière section de ce chapitre est consacrée à la présentation d'une étude comparative, le long du cycle de vie d'un logiciel, des différentes approches étudiées dans ce chapitre. Cette étude comparative nous a permis de déterminer les points forts et les points faibles de chaque approche.

Le deuxième chapitre expose *l'approche VUML* selon deux volets : le langage de modélisation et la génération de code. Le langage de modélisation de VUML est une extension d'UML intégrant la notion de vue/point de vue. Afin de concrétiser l'approche et montrer le passage à la phase de programmation, un patron générique d'implémentation est proposé pour favoriser la génération de code objet multi-cibles.

Le troisième chapitre est dédié à la sémantique (statique et dynamique) de VUML. Cette sémantique est décrite en utilisant le métamodèle VUML, des règles de bonne modélisation (well-formedness rules) et des descriptions textuelles précises.

Le quatrième chapitre présente les aspects applicatifs de ce travail. Il décrit premièrement la réalisation de l'outil support à VUML. Cet outil est implémenté en adaptant l'atelier *Objecteering/UML* au moyen de la technique des profils. En deuxième lieu, nous montrons en guise d'exemple l'utilisation de cet outil dans la modélisation d'un système d'enseignement à distance.

Le dernier chapitre a pour but de présenter une démarche associée à VUML dans le contexte de MDA (Model Driven Architecture). Après une présentation synthétique de l'approche MDA, ce chapitre décrit les grandes lignes d'une démarche associée à VUML dirigée par les modèles.

Enfin, nous concluons en récapitulant les points forts de VUML mais aussi en reconnaissant ses limites. Nous indiquons également quelles sont nos voies de recherche actuelles et futures.

Chapitre I

Etat de l’art

I.1. Introduction

Aujourd’hui, les entreprises doivent évoluer dans un environnement complexe, évolutif, dominé par une forte concurrence. La flexibilité, la puissance d'expression et les possibilités de réaction et d'adaptation aux besoins des clients restent des défis pour toute entreprise et suscitent des efforts de recherche dans les différents domaines des sciences de l'information.

Pour produire (ou faire) il faut d'abord modéliser "*car l'ingéniosité (modélisatrice) a été donnée à l'homme pour savoir, c'est-à-dire pour faire"* (G.B. Vico)[1]. Le domaine du génie logiciel a donc connu l'effervescence de plusieurs approches de modélisation et de production de système logiciel. Avec des objectifs convergents, les concepts mis en avant par chacune de ces approches sont souvent fondés sur la conjonction acteur/information, car la subjectivité, les facteurs endogènes, culturels, économiques, etc. ne peuvent pas être formalisés. Donc il faut inclure l’acteur dans l’action.

L’utilisation de l’approche orientée objet dans la modélisation des systèmes complexes a apporté au domaine du génie logiciel de nombreuses améliorations. En effet, la technologie objet permet d'améliorer la productivité, la fiabilité et la réutilisabilité des logiciels. Cependant, elle a tendance à ne pas intégrer l’acteur et sa vision par rapport au système ou à l'intégrer dans une étape tardive. L’intégration de la notion de vue/point de vue dans l’approche objet permet de prendre en compte les acteurs du système dès les premières étapes d'analyse et de conception. De ce fait, elle permet aux différents experts de se focaliser sur leur domaine d'expertise et de projeter leurs systèmes de valeurs sur le système.

Dans cette introduction, nous abordons tout d’abord la problématique de la modélisation des systèmes complexes (section I.1.1). Les limites de l'approche objet pour la modélisation des systèmes complexes sont présentées dans la section I.1.2. La section I.1.3 décrit l'apport de l'approche multivues pour pallier ces limites.

I.1.1. Problématique de la modélisation des systèmes complexes

Dans son livre "*La modélisation des systèmes complexes*", Jean-Louis Le Moigne (Le Moigne, 1990) définit un système complexe comme "*un système que l’on tient pour irréductible à un modèle*

---

[1] Pris d’une note de lecture rédigée par Jean-Louis Le Moigne et Magali Orillard sur leur ouvrage **:** "L'intelligence stratégique de la complexité"**,** Revue Internationale de Systémique, vol. 9, n-2, 1995 (Ed. Afcet-Dunod).

fini quelle que soit sa taille, le nombre de ses composants, l'intensité de leur interaction. Pour un observateur, il est complexe parce qu'il tient pour certain l'imprévisibilité potentielle des comportements. La complexité est une propriété attribuée au phénomène observé par l'acteur du fait des représentations qu'il s'en fait.".

La complexité d'un système est due selon Le Moigne à deux principales dimensions :
La dimension temporelle : c'est le fait qu'un système complexe soit toujours imprévisible dans le temps, indéterministe (non réductible à un système à états fini), et le même observateur, avec un même angle de vision mais à des instants différents *(ou sessions)* peut se faire des représentations différentes du système observé. On parle alors de la *dynamicité* ou de la *métamorphose* du système.
La dimension spatiale : selon le lieu d'observation et l'observateur *(ou Profil)*, la quantité *d'information* qu'a l'observateur sur le système est différente. En effet, imprégné dans un système de valeur, de cultures et de facteurs endogènes, chaque observateur comprend, pense et agit vis à vis d'un système de manière différente. Le rapport entre la quantité d'information qu'a l'observateur sur le système et la quantité d'informations réellement contenue dans ce système peut varier de 0 *(système chaotique)* à 1 *(système simple).*

Cette complexité est donc appréhendée par l'observateur (vue/temps) qui s'y intéresse. Le modélisateur qui essaie de rendre intelligible l'information réelle du système, représente artificiellement cette dernière selon la conception et la vue qu'il construit et reçoit vis à vis de ce système. Un système peut donc être représenté par un ensemble de variétés possibles selon la projection de l'acteur, la finalité du système, etc.

Chaque système complexe est donc par nature *multivues,* et chaque interaction observateur/système peut être considérée comme une *expérience interactive.* Nous pouvons ainsi dire qu'à chaque espace-temps donné de l'observateur, est associé un espace-temps *(vues)* du système.

Le développement et la modélisation de tels systèmes ont longtemps été handicapés par l'absence de véritable définition des concepts d'information qui implique ces facteurs induits par les systèmes complexes. Une première approche consistait en une décomposition des systèmes sous forme de modèles partiels qui constituent les différents comportements possibles du système ou les variétés du système. Ces variétés sont le résultat de la conjonction des concepts d'Environnement (caractère multivues de l'observateur), de finalités (à chaque acteur correspond une finalité du système), de fonctionnalités (le système joue plusieurs rôles selon le besoin de l'acteur) et de transformations (le système est sujet à des métamorphoses durant son cycle de vie). Ainsi une variété se décrit selon un environnement pour des projets (finalités) fonctionnant et se transformant.

Ces variétés nécessitent des moyens de communication, de coordination et de cohérence pour gérer les inter-relations qui lient les différentes projections des différents acteurs et les éventuels enchevêtrements.

La modélisation multi-modèles par l'approche objet s'est révélée féconde pour une meilleure structuration et réutilisation des systèmes complexes mais elle ne permet pas la gestion des incohérences et la redondance de données entre les différentes variétés du système.

I.1.2. Limites de l’approche objet pour la modélisation des systèmes complexes

Par rapport aux approches dites *classiques*, il est indéniable que l'approche objet et les concepts associés (encapsulation, héritage, polymorphisme) permettent la construction de modèles plus simples à comprendre, à valider, à faire évoluer et à réutiliser.

L'approche objet fournit un processus de conception guidé par les données. Elle se fonde essentiellement sur (Rumbaugh et al., 1991) :

- **l'identité** : tout objet a une identité qui est uniforme et indépendante du contenu ;
- **la classification** : tout objet est une instance d'une classe, qui définit sa structure et son comportement ;
- **le polymorphisme** : un traitement peut avoir des interprétations différentes selon la classe de l'objet auquel on l'applique ;
- **l'héritage** : il permet de promouvoir l'extensibilité et la réutilisabilité des classes en les liant par une hiérarchie taxonomique.

Ces fondements de l'approche objet semblent bien adaptés à l'identification et à la structuration du monde réel. Mais, "Le monde réel n'est ni plat ni séquentiel, mais au contraire multidimensionnel et hautement parallèle" Grady Booch (Booch, 1991).

Cette complexité du monde réel, son caractère multi-dimentionnels (comme montré dans la section précédente) donnent lieu tout naturellement à une approche multi-modèles. Cette pratique qui consiste à construire plusieurs modèles partiels d’un système engendre inévitablement des problèmes d’incohérence et de redondance de données entre les modèles partiels. En effet, vu la complexité de ces systèmes (par exemple un satellite), leur modélisation nécessite un ensemble parfois important de concepteurs, spécialistes et experts. La modélisation est menée en découpant le système en plusieurs modèles partiels autonomes afin d'éviter la conception d'un grand système. Ceci entraîne des problèmes de gestion de la cohérence entre ces modèles partiels, car ceux-ci ne sont jamais indépendants, puisqu’ils contiennent des informations souvent fonctionnellement liées. Le travail de fusion des modèles partiels, qui est mené itérativement pour produire une modélisation cohérente, est lui-même un processus complexe qui ne donne pas toujours satisfaction et qui peut nécessiter des délais très importants. De même, ces modèles partiels posent aussi des difficultés pour les maintenir et les réutiliser.

Nous pouvons donc constater que la technologie objet ne permet pas en tant que telle de résoudre les problèmes d'incohérence et de redondance entre les modèles partiels. L’intégration de la notion de vue/point de vue dans l’approche objet semble être un moyen pertinent pour résoudre ces problèmes de modélisation de systèmes complexes. Le principe est d’élaborer un seul modèle partageable, accessible selon plusieurs visions, ce qui permet de surmonter les problèmes de redondance et de cohérence. Il s’agit donc en fin de compte de *"rétablir les articulations entre ce qui est disjoint, essayer de comprendre la multidimensionalité, penser avec la singularité..."*, selon les termes de E. Morin.

I.1.3. Apports de l'approche multivues

L’approche multivues a été introduite dans différents domaines de l’informatique et de la modélisation des systèmes. Ainsi, elle a fait l’objet de plusieurs travaux dans les bases de données (Abiteboul et al. 1991, Debrauwer 1998), la représentation des connaissances avec les systèmes

TROPES (Marino et al. 1990, Marino 1991, Marino 1993) et SHOOD (Rieu et al. 1992a, Rieu et al. 1992b), des outils de génie logiciel tels que des serveurs de vues (Cueignet et al., 1992), des langages à objets (PIE (Goldstein et al, 1980), Janus (Habermann et al., 1988), YAFOOL (Ducourneau et al., 1989), OBJLOG (Dugerdil, 1988), ROME (Carré et al., 1990a), FROME (Dekker et al. 1992, Dekker 1992, Dekker 1994, etc.), VBOOL (Marcaillou 1995, Nassar 1999) et dans la modélisation du processus de développement (Finkelstein et al. 1990, Coulette et al. 1996, Kriouile 1995, Motsching-Pitrik 2000, etc.).

L'introduction de la notion de point de vue dans la modélisation orientée objet des systèmes complexes consiste en l'élaboration d'un modèle unique partageable accessible suivant plusieurs points de vue. L'intérêt de cette approche apparaît au niveau de la cohérence des données, de la suppression de certaines redondances, de l'enrichissement de l'approche multi-modèles et de la définition des droits d'accès (Marcaillou, 1995).

Cohérence

Contrairement à l'approche multi-modèles, l'approche multivues permet de surmonter les problèmes d'incohérence dus au partage des données entre les modèles partiels. En effet, l'approche multivues se base sur l'élaboration d'un modèle unique partageable accessible selon plusieurs points de vue, ce qui permet de centraliser les données du modèle. L'avantage majeur de l'utilisation d'un modèle unique est que chaque utilisateur d'un sous-modèle voit les modifications apportées répercutées dans les autres sous-modèles. Par conséquent, les problèmes d'incohérence qui constituent l'inconvénient majeur de l'approche multi-modèles sont évités.

Le principe de l'approche multivues qui vise à utiliser un modèle unique est très proche de celle du "modèle partagé" présenté dans (Fowler, 1994) et (Rumbaugh, 1994). En effet les deux approches tentent d'éviter la redondance et l'incohérence des données en fournissant un modèle commun avec des accès sélectifs.

Enrichissement de l'approche multi-modèles

La notion de point de vue associée aux utilisateurs donne la possibilité de voir le modèle selon plusieurs points de vue. Ceci revient à donner des "éclairages" différents sur la même information, ce qui permet à l'utilisateur de se focaliser sur un sous-modèle du modèle global spécifique à son besoin. Donc la notion de point de vue induit indirectement un enrichissement de l'approche multi-modèles.

Droits d'accès

Les utilisateurs d'un système n'ont pas les mêmes besoins et les mêmes responsabilités, ce qui met en évidence la nécessité de définir des droits d'accès aux informations du modèle. L'intégration de l'approche multivues dans la modélisation par objet permet de prendre en considération cette exigence. En effet, les points de vue se traduisent par la définition de droits d'accès sur le modèle. Ils permettent donc de masquer des vues à certains utilisateurs.

Centralisation de la connaissance

Dans un modèle, certaines connaissances doivent être partagées entre des utilisateurs différents. Pour assurer ce partage, l'approche multi-modèles procède généralement par une duplication de ces connaissances dans les différents sous-modèles. Cependant, cette répartition des informations implique la nécessité de prendre en charge la mise à jour de ces informations pour garder leur cohérence. Afin de surmonter ces problèmes, l'approche multivues utilise un modèle unique accessible selon plusieurs points de vue.

Evolutivité

Un intérêt important de l'approche multivues est la possibilité de masquer puis de retrouver des points de vue à des instants donnés. Ceci veut dire qu'il faut conserver les modèles d'analyse tout au long du cycle de vie d'un système mais sans qu'ils soient actifs en permanence. Donc les points de vue permettent de répondre au besoin d'évolutivité temporelle. Ils offrent aussi la possibilité de faire évoluer les droits d'accès des utilisateurs du modèle.

I.1.4. Contenu du chapitre

Dans le reste de ce chapitre, nous présentons le projet VBOOM (section I.2). Ensuite nous décrivons rapidement quelques travaux traitant la notion de vue/point de vue en développement des systèmes : représentation des connaissances (section I.3.1), génie logiciel (section I.3.2). La section I.3.3 fait l'objet d'une présentation concernant les paradigmes de programmation qui s'apparentent aux points de vues, à savoir : programmation par sujets, programmation par aspects, programmation par vues, programmation par objets en contexte, objets morcelés dans les langages à prototypes. La section I.3.4 décrit des travaux représentatifs concernant la notion de vue/point de vue en modélisation.

Enfin, nous concluons ce chapitre en dressant un bilan comparatif des différentes approches étudiées.

Nous tenons à signaler que nous avons aussi étudié l'ingénierie dirigée par les modèles et en particulier l'approche MDA (Model Driven Architecture) (OMG, 2003d). En effet, l'élaboration d'une conception VUML s'apparente à une transformation de modèles UML de niveau PIM (Platform Independent Model). De ce fait, nous avons placé l'étude de l'approche MDA dans le chapitre V consacré à l'aspect méthodologique de notre travail.

I.2. Projet VBOOM

Le projet VBOOM[2] (View Based Object Oriented Methodology) a donné lieu à plusieurs travaux de recherche. Ces travaux ont visé l'introduction du concept de vue et point de vue dans le cadre d'une modélisation par objets. L'objectif était d'apporter une plus grande souplesse, flexibilité et rigueur dans la modélisation à objets dans un contexte de Génie Logiciel. Le modélisateur d'un système a priori complexe (par exemple un satellite) doit pouvoir définir des vues multiples sur ce système, puis accéder à l'une ou à plusieurs de ces vues selon ses besoins particuliers. Pour cela, un nouveau concept « la classe flexible », et une nouvelle relation appelée « visibilité », intégrés au sein d'une extension d'Eiffel (Meyer,1992) appelée VBOOL – View Based Object-Oriented Language – (Marcaillou, 1995), ont été définis. Dans le même contexte, une méthode d'analyse et de conception par objet VBOOM – View Based Object Oriented Method – (Kriouile, 1995) supportant les concepts de vue et point de vue a été élaborée. Cette méthode permet de modéliser un système selon les points de vue de ses différents utilisateurs. Ceci conduit à élaborer un certain nombre de modèles partiels appelés « modèles visuels ». Une fois ces modèles réalisés – éventuellement de façon décentralisée – une étape de fusion de ces modèles est proposée par la méthode VBOOM. Cette fusion porte sur les interfaces des classes composant les modèles partiels. Elle a été formalisée comme une loi de composition

[2] Le Projet VBOOM a été lancé en 1992, il s'est déroulé en partenariat avec le laboratoire mixte ARAMIIHS de Toulouse et le laboratoire d'Informatique de l'ENSIAS de Rabat

interne sur l'ensemble des interfaces de classes. Un prototype d'environnement supportant la méthode, appelé VBTOOL (Marzak 1997, Hair 1997, Hair et al 1998), a été réalisé.

VBOOM a été appliquée à la modélisation de plusieurs systèmes complexes dont un sous-ensemble de la case ARIANE IV (Marcaillou, 1995) et un système de gestion de l'interaction transport/environnement au Maroc (El Asri et al., 1998).

Dans la suite de cette section, nous allons présenter le langage VBOOL et la méthode VBOOM ainsi que son atelier support VBTOOL. Ensuite, nous décrirons les limites de l'approche développée au sein du projet VBOOM.

I.2.1. Le langage VBOOL (View Based Object Oriented Language)

Afin de satisfaire les besoins de modélisation des systèmes complexes dans un contexte de génie logiciel, Marcaillou et al. ont étendu l'approche orientée objet en intégrant les points de vue dans la modélisation par objet. L'implantation de la notion de vue et de point de vue est réalisée en définissant une nouvelle relation appelée *visibilit*é. Celle-ci, proche de l'héritage, permet de sélectionner les informations contenues dans une classe et de les filtrer vers les classes dérivées, selon un principe analogue à la définition des statuts d'exportation en délégation (Marcaillou et al., 1996). Pour supporter la visibilité, un langage fortement typé appelé VBOOL (View Based Object Oriented Language) a été défini.

Dans la suite de cette section nous présentons les notions de base de VBOOL - vue/point de vue et relation de visibilité - puis nous abordons la compatibilité de cette nouvelle relation avec les relations objet déjà existantes (héritage, agrégation, composition). Les notions introduites dans cette section sont illustrées par l'exemple de la modélisation d'une *station de travail*. Les différents utilisateurs potentiels d'une station sont : l'étudiant, l'ingénieur système, l'administrateur, etc. Ces divers utilisateurs n'ont évidemment pas les mêmes besoins et ne doivent donc pas avoir les mêmes droits d'accès aux informations. Par conséquent leurs points de vue sur la station seront différents les uns des autres. La notation utilisée dans cette section est spécifique à la méthode VBOOM (inspirée de la notation de la méthode BON (Walden et al., 1995)).

I.2.1.1. Relation de visibilité

Dans VBOOL (Marcaillou, 1995), les vues et les points de vue sont définis informellement comme suit :

- **Une vue** est une abstraction partielle du modèle, correspondant donc à un sous-modèle.
- **Un point de vue** est la vision qu'a un utilisateur du modèle et correspond à la combinaison de plusieurs vues, éventuellement réduite à une seule.

Le principe de base de l'utilisation de la visibilité est lié à sa sémantique que l'on peut exprimer par '*est vu comme'*. Conceptuellement, la relation de visibilité est une duplication virtuelle dans une classe des informations provenant de ses vues. Cette relation est très proche de la relation d'héritage dont la sémantique est "est un". Cependant, il y a une différence importante qui se manifeste dans la possibilité de sélectionner des informations et d'appliquer des mécanismes de filtrage. Cette nouvelle relation constitue donc une extension de l'héritage : c'est un héritage multiple sélectif qui apporte plus de souplesse. Ce type d'héritage permet à une classe de bénéficier de toute l'information contenue dans ses vues et de la filtrer en fonction de l'utilisation qui en est faite, c'est-à-dire en fonction des points de vue spécifiés (Marcaillou, 1995).

N'importe quelle classe est susceptible de devenir une vue puisqu'une classe quelconque A peut établir une relation de visibilité vers une classe B qui devient alors une vue de A.

Considérons la classe *ORDINATEUR* dans notre exemple (cf. figure 1) ; elle devient la vue *ORDINATEUR* pour la classe *STATION* par une déclaration d'un lien de visibilité dans cette classe.

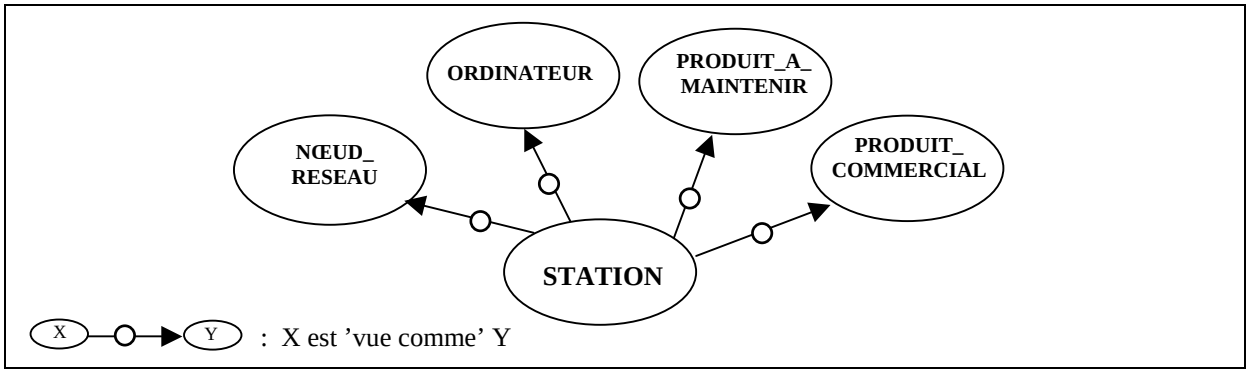

Figure 1 – *Expression de la relation de visibilité*

I.2.1.2. Quelques éléments syntaxiques de VBOOL

Le langage VBOOL a été élaboré pour supporter la relation de visibilité et les concepts qui lui sont liés. La construction d'un nouveau langage a été justifiée par le fait qu'il n'y avait pas de langage satisfaisant sur le marché, étant donné que si certains étaient fortement typés ou offraient une possibilité d'évolution dynamique, aucun ne supportait simultanément les deux fonctionnalités, c'est-à-dire la relation de visibilité telle que définie dans (Marcaillou, 1995).

VBOOL est une extension du langage Eiffel (Meyer, 92) qui supporte donc tous les mécanismes objet : héritage, délégation, encapsulation, polymorphisme, généricité, contrats, traitement des exceptions.

Dans la suite de cette section nous décrivons les concepts et les mécanismes principaux de VBOOL. D'autres concepts et mécanismes de ce langage sont présentés dans (Marcaillou 1995, Marcaillou et al. 1996).

Concept de classe flexible, déclaration des vues

Une classe flexible est une classe dans laquelle est déclaré un ensemble de vues constitué d'au moins deux éléments. Elle constitue une potentialité de vues qui seront précisées au moment du choix d'un point de vue particulier, lors de la déclaration d'une instance de la classe flexible.

- **Structure d'une classe flexible**

Une classe flexible est constituée de trois parties (cf. figure 2) :

- Une partie commune (publique) qui contient toutes les primitives communes à toutes les instances, et leurs statuts d'exportation en délégation. Elle constitue donc la partie d'une classe Eiffel accessible par ses instances.

- Une partie privée qui a accès aux informations provenant de toutes les vues de la classe flexible. Cette partie n'est accessible que depuis les autres parties de la classe.

- Une partie visible qui gère tout ce qui concerne la visibilité. Elle décrit les vues de la classe en spécifiant celles qui sont en exclusion mutuelle, puis les statuts d'exportation en visibilité des primitives de la classe. Cette partie permet de définir le comportement adapté aux points de vue définis sur la classe flexible. Il est aussi possible, lorsqu'on déclare une vue d'une classe flexible, d'introduire une clause de renommage ou de redéfinition, comme lors de l'héritage en Eiffel.

```
flexible class A                                    -- déclaration de la classe flexible

inherit B                                           -- clause d'héritage
        rename meth1 as meth2                       -- renommage d'une routine héritée
        redefine meth3                              -- redéfinition d'une routine héritée
        end;
        C
        rename meth3 as meth4                       -- renommage d'une routine héritée
        end;

creation   make

                                -- ZONE PUBLIQUE

feature {ANY}                                       -- exportation en délégation vers tout client

        make is do...end;
        meth3  is do...end;

                                -- ZONE PRIVEE

feature {NON}                                       -- primitives privées

        methA2 is do...end;
        methA3 is do    meth1_V1;   end;            -- appel d'une méthode de la vue V1

                                -- ZONE VISIBLE

seen_as  V1                                         -- déclaration de la vue V1

        rename  meth2_V1 as meth_V1                 -- renommage de meth2_V1 provenant de V1
        redefine meth3_V1                           -- redéfinition de la routine meth3_V1 de V1
        end;

seen_as V2                                          -- déclaration de la vue V2
....
end  -- class A
```

Figure 2 – *Classe flexible avec 2 vues en VBOOL*

- **Expression en VBOOL d'une classe flexible**

La déclaration d'une classe flexible VBOOL se fait en utilisant le mot clé '*flexible*' dans l'entête de la classe. Les vues d'une classe flexible sont déclarées dans des clauses spécifiées par le mot clé '*seen_as*'. L'exemple de la figure 2 ci-dessus illustre la déclaration en VBOOL d'une classe flexible A avec deux vues V1 et V2.

Déclaration d'un point de vue

La déclaration d'un point de vue sur une classe flexible consiste à préciser, lors de la déclaration d'une instance et de son type statique, les vues constituant le point de vue adapté.

Syntaxe VBOOL : *objet : Nom_classe_flexible (vue1, vue2,...);*

Exemple : Le point de vue d'un étudiant (PVE) comporte les vues *ORDINATEUR* et *NŒUD_RESEAU*.

PVE : STATION(ORDINATEUR, NŒUD_RESEAU) ;

Le point de vue de l'administrateur (PVA) comporte les vues : *ORDINATEUR, NŒUD_RESEAU, PRODUIT_A_MAINTENIR*.

PVA : STATION(ORDINATEUR, NŒUD_RESEAU, PRODUIT_A_MAINTENIR) ;

Exclusion mutuelle de vues

Les vues en exclusion mutuelle sont des vues qui ne peuvent jamais être spécifiées simultanément dans un même point de vue. Considérons l'exemple d'une station : l'administrateur et l'étudiant ont le droit d'accéder aux informations de la vue *ORDINATEUR*. Cependant, seul l'administrateur a l'autorisation de changer la configuration d'une station. Pour exprimer cela, nous pouvons créer par héritage deux vues héritant de *ORDINATEUR*, les vues *ORDINATEUR_CONFIGURE* et *ORDINATEUR_CONFIGURABLE*. Ces deux vues étant en exclusion mutuelle (cf. figure 3), il devient possible d'exprimer que l'administrateur a accès à *ORDINATEUR_CONFIGURABLE* et que l'étudiant a accès à *ORDINATEUR_CONFIGURE*.

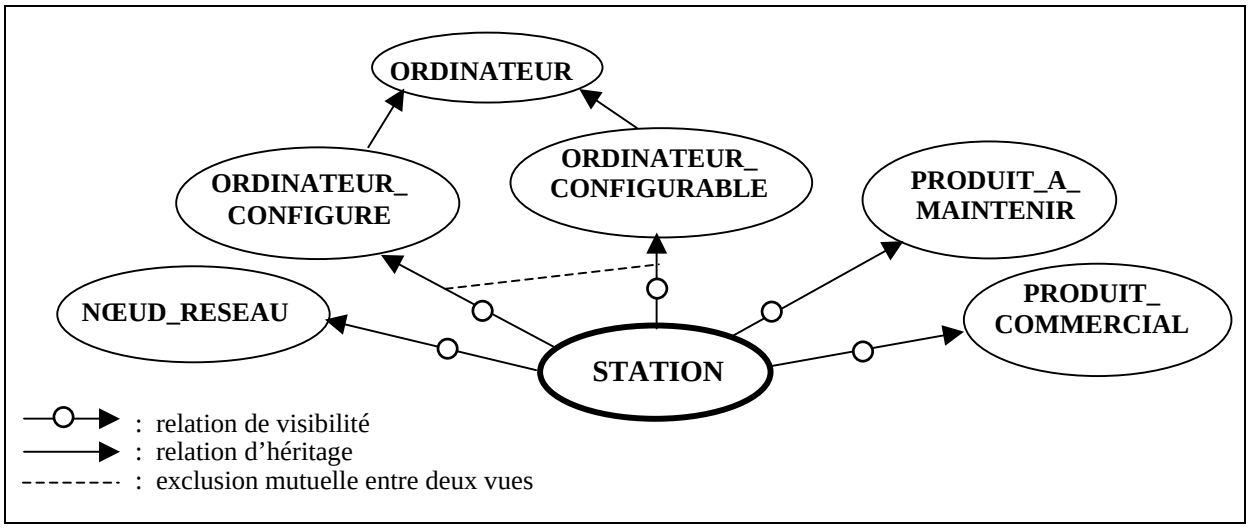

Figure 3 – *Exemple de vues exclusives de la classe STATION*

Expression en VBOOL de l'exclusion mutuelle :

Les vues exclusives sont déclarées dans une clause '*seen_as*' de la classe flexible. L'exclusion mutuelle est spécifiée par le symbole '/' (cf. figure 4).

```
flexible class STATION
…
seen_as  NŒUD_RESEAU
seen_as ORDINATEUR_CONFIGURABLE  /  ORDINATEUR_CONFIGURE  ;        -- exclusion mutuelle
…
end
```

Figure 4 – *Définition de vues exclusives dans la classe flexible STATION*

Dérivation par visibilité

L'héritage en VBOOL est une extension de celui défini en Eiffel. Il constitue toujours une duplication virtuelle des informations. La dérivation par visibilité consiste à préciser un point de vue sur une classe flexible C2 dans une clause *seen_as* d'une classe C1. Ce point de vue devient donc lui-même une vue de la classe C1.

Dans la figure 5 la classe flexible *STATION_PORTABLE* possède une vue dérivée W, définie par visibilité sur la classe flexible *STATION*. L'exemple de la figure 6 montre comment se fait la déclaration des vues dérivées en VBOOL.

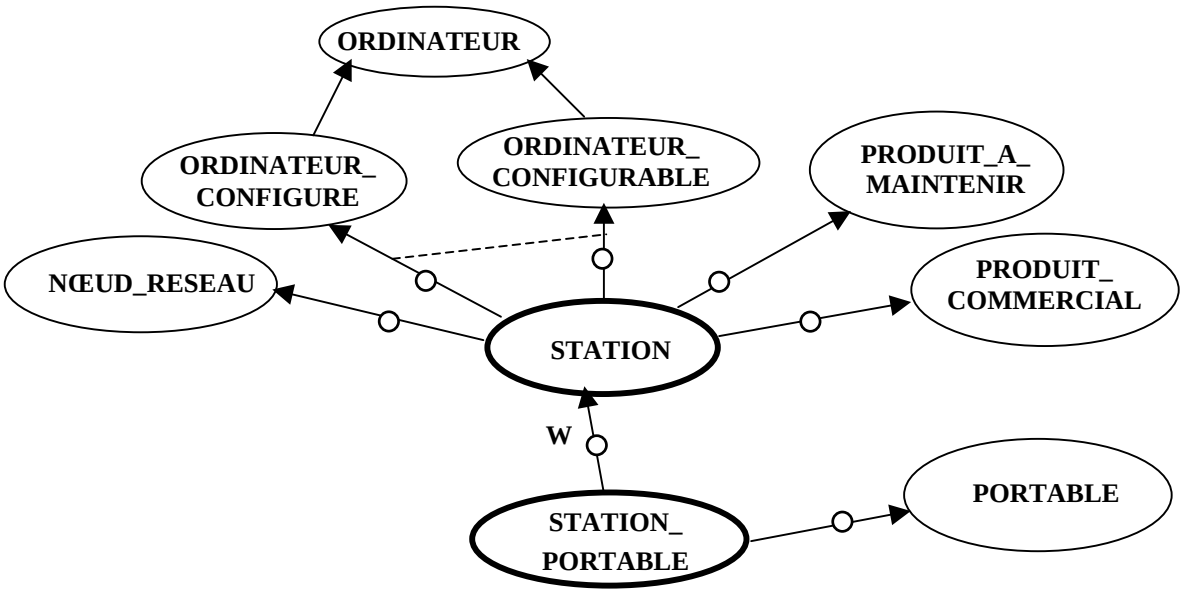

Figure 5 – *Dérivation par visibilité*

```
flexible class STATION_PORTABLE
...
seen_as  PORTABLE
seen_as W= STATION(ORDINATEUR_CONFIGURABLE, PRODUIT_A_MAINTENIR)
...
end   -- class STATION_PORTABLE
```

Figure 6 – *Dérivation d'une classe par visibilité*

Exportation en visibilité

VBOOL étant une extension d'Eiffel, tout ce qui concerne les statuts d'exportation dans Eiffel reste valable dans VBOOL. Les primitives provenant des vues d'une classe flexible sont utilisables dans la partie privée et la partie visible de la classe flexible, et bien sûr dans ses descendantes par héritage. Une instance d'une classe flexible ne peut invoquer une primitive provenant d'une vue que si sa classe d'instanciation l'y autorise, donc que si cette vue fait explicitement partie du point de vue représenté par cette instance.

VBOOL permet aussi d'utiliser des primitives restreintes aux instances ayant un point de vue spécifique sur cette classe. Ces primitives doivent être déclarées dans la partie visible de la classe flexible. Cette déclaration se fait en utilisant un mécanisme spécifique à VBOOL, appelé "*mécanisme d'exportation sélective*". Dans VBOOL le mécanisme d'exportation sélective est réalisé est introduit à travers le mot clé '*view_feature'*.

Syntaxe VBOOL : *view_feature combinaison_de_vues_concernées*

L'exemple de la figure 7 ci-dessous montre que toute instance de la classe flexible *STATION* pourra invoquer la routine ***update*** à condition qu'elle ait exactement le point de vue suivant :
NŒUD_RESEAU, PRODUIT_A_MAINTENIR

Evolution dynamique des points de vue

Afin de permettre l'évolution dynamique des points de vue, VBOOL offre la possibilité de faire varier l'ensemble des vues d'un point de vue donné. Pour réaliser ce mécanisme, VBOOL propose un ensemble d'opérations qui peuvent être appliquées à une instance d'une classe flexible. Ces opérations sont :

establish_view ("V1","V2",…,"Vn") : active les vues V1 à Vn ;
suppress_view ("V1", "V2",…,"Vm") : désactive les vues V1 à Vm ;
exchange_view ("Vn","Vm") : remplace la vue Vn par Vm.

```
flexible class STATION
-- Zone publique
feature {ANY}
…
-- Zone privée
feature {NONE}
…
-- Zone visible
seen_as NŒUD_RESEAU
seen_as ORDINATEUR_CONFIGURABLE / ORDINATEUR_CONFIGURE
seen_as PRODUIT_A_MAINTENIR
seen_as PRODUIT_COMMERCIAL

view_feature NŒUD_RESEAU, PRODUIT_A_MAINTENIR
display is do … end ;
update is do ...end ;
…
view_feature ORDINATEUR, NŒUD_RESEAU
display is do … end ;
…
end -- class STATION
```

Figure 7 – *La clause view_feature dans une classe flexible*

Mise en cohérence des vues

Le maintien de la cohérence des informations contenues dans le modèle est une tâche essentielle que VBOOL permet de garantir, notamment dans le cas des vues exclusives. La répercussion des modifications effectuées sur un point de vue et mettant en cause une vue, qui ne fait pas partie de ce point de vue, se fait obligatoirement en utilisant la classe flexible. En effet, les vues d'une classe flexible s'ignorent entre elles, même si elles peuvent être fonctionnellement dépendantes. Du fait que les routines déclarées dans la partie privée d'une classe flexible ont le droit de mettre à jour les informations provenant des vues, le programmeur doit utiliser cette partie pour définir les routines permettant la répercussion des modifications entre les vues.

I.2.2. La méthode VBOOM (View Based Oriented Object Method)

Afin d'améliorer la productivité, la fiabilité et la réutilisation des logiciels, plusieurs méthodes d'analyse et de conception orientée objet ont été élaborées. Parmi ces méthodes, nous citons OOA/OOD (Coad et al., 1991), OMT (Rumbaugh et al., 1991), CLASS/RELATION (Desfray, 1992), OOD (Booch, 1994), OOSE (Jacobson et al., 1993), BON (Waldén et al., 1995), etc. Certaines de ces méthodes (telle que OOSE) utilisent des notions proches de la notion de points de vue. Cependant, aucune d'entre elles ne permet de gérer les vues et les points de vue durant toute la phase de modélisation jusqu'à l'obtention du code du système. La méthode VBOOM (Kriouile, 1995) a été élaborée pour combler ce manque. C'est une méthode d'analyse et de conception orientée objet qui intègre l'approche multivues dans la modélisation par objet. Cette méthode est une extension de la méthode B.O.N (Business Object Notation) développée par J.M. Nerson (Waldén et al., 1995) et cible le langage VBOOL. VBOOM possède un atelier de génie logiciel appelé VBTOOL (Hair 1997, Marzak 1997, Hair et al. 1998). Il permet de mener une analyse/conception par points de vue.

I.2.2.1. Modèle de développement supporté par VBOOM

La figure 8 ci-dessous présente le modèle de développement par points de vue (Coulette et al., 1996) supporté par la méthode VBOOM. C'est un processus itératif et éventuellement récursif de trois phases, chacune d'elles étant composée de trois étapes (Kriouile, 1995). La première phase est une phase d'analyse qui a pour objectif l'identification des différents constituants du modèle global. Ces constituants forment le dictionnaire initial qui sera utilisé dans la conception des modèles visuels lors de la deuxième phase. Cette dernière est une phase décentralisée permettant de concevoir les différents modèles visuels (modèles partiels représentant les différents points de vue). La troisième phase a pour but de fusionner les modèles visuels en élaborant un modèle unique.

Après avoir obtenu la première version du modèle global, le processus entre dans sa phase d'exploitation durant laquelle nous pouvons poursuivre le développement pour implanter les différentes classes du système. Cette implantation se fait selon un modèle de développement objet tel que celui dit par "grappes" (Meyer 1995, Waldén et al. 1995). Nous pouvons aussi extraire des modèles visuels afin de faire des mises à jour, des simulations, etc. Ces modifications seront prises en compte dans le modèle global en fusionnant de nouveau les modèles visuels modifiés.

I.2.2.2. La démarche de VBOOM

Les méthodes d'analyse et de conception sont généralement fondées sur une modélisation et une démarche. La modélisation permet de définir un ou plusieurs modèles et un ensemble de règles à

respecter, alors que la démarche définit un processus de mise en œuvre. VBOOM respecte ce principe en proposant une modélisation couvrant les aspects statiques et dynamiques pour décrire le système et son comportement. La démarche de VBOOM intègre l'approche hiérarchique par points de vue présentée dans la figure 8. Conformément à cette approche, la démarche de VBOOM est un processus itératif de trois phases, chacune étant constituée de trois étapes. Cette démarche concernant un seul niveau de point de vue, il suffit de réitérer les phases proposées pour traiter plusieurs niveaux de points de vue. Dans la suite, nous présentons succinctement ces trois phases résumées dans la figure 9 (Kriouile, 1995).

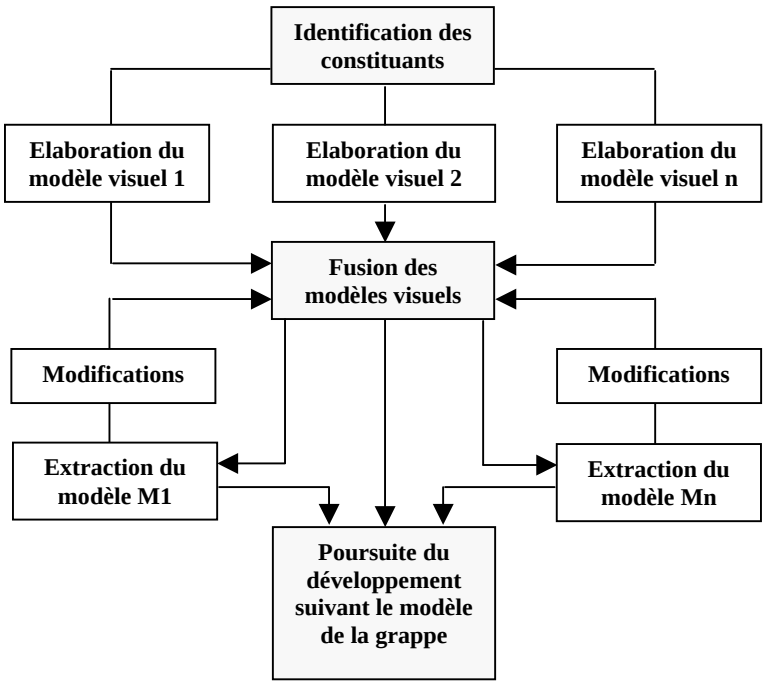

Figure 8 – *Représentation d'un niveau du modèle de développement par points de vue*

- **Phase "identification des constituants"** : c'est une phase centralisée qui consiste à identifier les différents constituants du domaine du problème (dictionnaire des constituants) en identifiant les acteurs et leurs besoins, les points de vue, les classes, les classes flexibles et leurs vues.

- **Phase "élaboration des modèles visuels"** : elle consiste à découper l'espace de la solution du domaine du problème en sous-domaines plus simples associés aux différents points de vue des acteurs potentiels du système, et à concevoir les modèles partiels correspondants (les modèles visuels). La conception de ces modèles visuels est une tâche décentralisée.

- **Phase "fusion des modèles visuels"** (Kriouile 1995, El Asri et al. 1996) : c'est une phase centralisée qui a pour but la fusion des modèles visuels obtenus à l'issue de la phase précédente pour élaborer le modèle global du système. Bien sûr cette phase doit prendre en charge la résolution des conflits qui peuvent exister entre les différents modèles visuels. En effet, la décentralisation de la phase de la conception qui est le point fort de la méthode VBOOM entraîne certains types de conflits qui doivent être résolus durant la fusion. Pour améliorer cette phase, des travaux ont été menés pour intégrer l'aspect coopératif dans la méthode VBOOM (Dhiba, 1999).

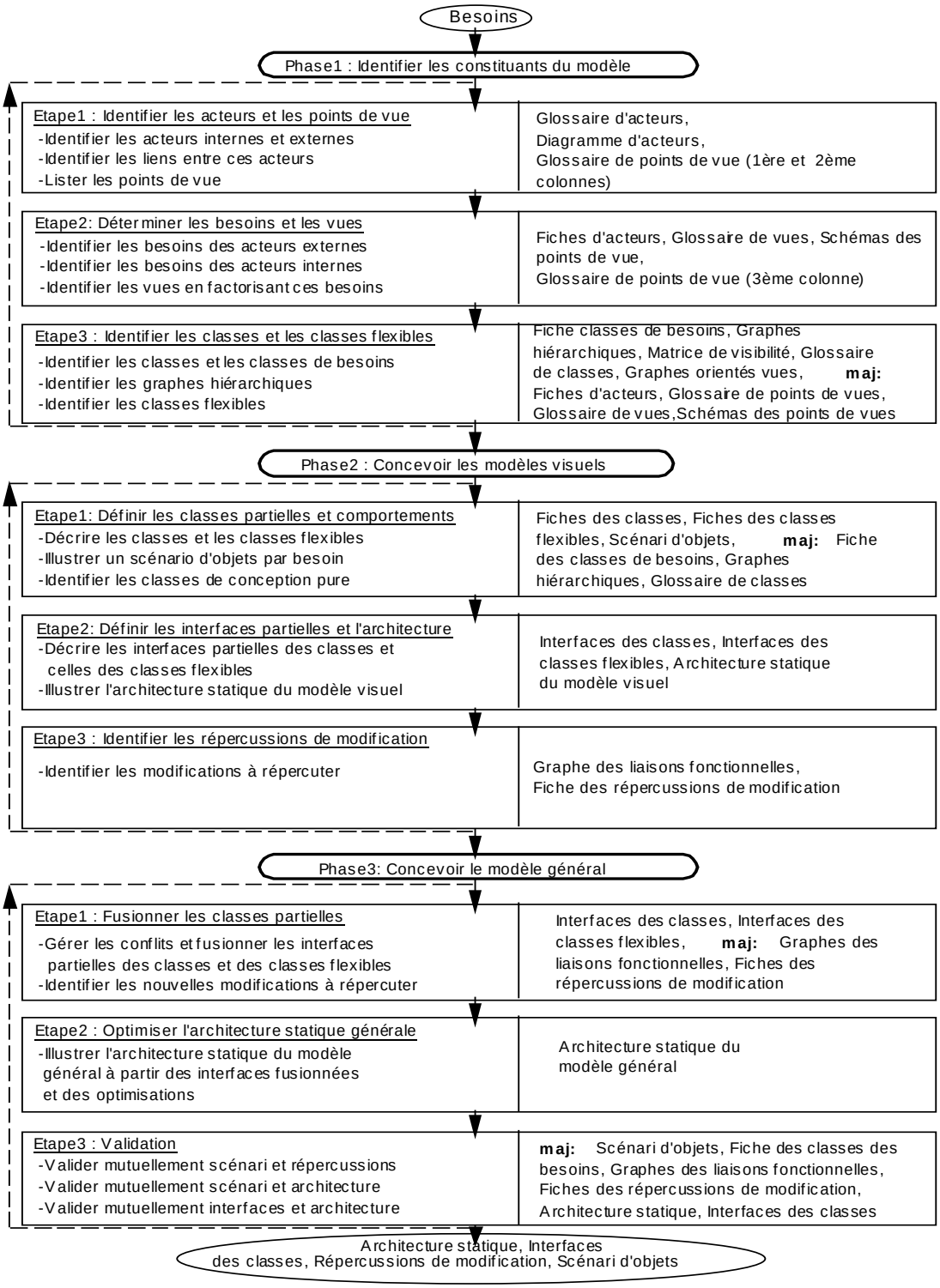

Figure 9 – *Démarche de VBOOM*

I.2.3. Limites de l'approche VBOOM

Les recherches menées au sein du projet VBOOM ont permis de disposer d'un environnement complet de modélisation par points de vues. Cependant, les expérimentations effectuées ont mis en évidence plusieurs limites de VBOOM ou de sa mise en œuvre :

- Subjectivité dans la définition des vues. En effet, lors de notre expérience d'analyse/conception de plusieurs systèmes (un sous-ensemble de la case ARIANE IV, un système de gestion de l'interaction transport/environnement, etc.) nous avons constaté que le processus d'identification des vues est délicat et fortement non déterministe.
- Complexité de modélisation avec VBOOM due à la dissociation des notions de vue et point de vue, et aux multiples combinaisons de vues possibles.
- Rigidité de l'héritage multiple pour prendre en compte l'évolution d'un modèle à base de vues. En effet, VBOOL (langage cible de VBOOM) est fondé sur l'héritage multiple qui supporte mal l'évolutivité du modèle. Ainsi, l'ajout d'une vue peut remettre en cause la classe flexible et donc les classes qui en dérivent.
- Formalisation et Implémentation délicates du polymorphisme en VBOOL.
- Non-conformité de la méthode VBOOM aux standards de l'OMG et aux langages objets du marché.
- Manque d'outils performants pour le support à VBOOM.

I.3. Notion de vue/point de vue en développement

Les concepts de vue et de point de vue ont été étudiés dans plusieurs domaines liés au traitement de l'information : bases de données, représentation des connaissances, analyse et conception, langages de programmation, outils de Génie logiciel, etc. Dans l'approche objet, ces concepts ont été employés sous différents noms : vue, point de vue, rôle, perspective, aspect, sujet, morceau, critère, etc. La notion de vue dans UML (OMG, 2003a) est un moyen offert au concepteur pour structurer une conception en fonction des aspects qu'il souhaite modéliser (cas d'utilisation, logique, composants, déploiement). Dans le domaine des bases de données, les vues sont exploitées par les langages d'interrogation comme des fonctions de sélection sur les données. Au niveau des langages de programmation par objet, les notions de vues ont également été expérimentées, sans que ne se dégage pour le moment de solution globale implémentée dans les principaux langages.

En ce qui concerne les méthodes d'analyse/conception par objet, il n'y a pas de méthodologie reconnue capable de supporter complètement la notion de point de vue.

Nous signalons l'existence d'une norme IEEE (la norme IEEE 1471) (Hilliard, 2000) qui prend en compte les points de vue lors des phases de définition de l'architecture d'un système logiciel. Cette recommandation propose un modèle conceptuel général pour décrire l'architecture d'un système. Chaque acteur (participant) du système a un ou plusieurs centres d'intérêt, chacun d'eux étant couvert par un ou plusieurs points de vue. Chaque point de vue est relié à une unique vue sur le système, chaque vue consistant en un ou plusieurs modèles. La dépendance entre les modèles d'une vue peut être explicitée à travers des relations de type UML. Cependant, la norme IEEE 1471 se situe à un niveau de granularité large, ne propose pas de langage de modélisation particulier, et n'offre pas une démarche pour trouver les points de vue.

Dans la suite de cette section, nous abordons l'utilisation de la notion de vue/point de vue dans les principaux domaines liés au développement du logiciel : représentation de connaissances, génie logiciel, langages de programmation et langages/méthodes de modélisation.

I.3.1. Vues et représentation des connaissances

- ROME (Carré 1989, Carré et al. 1990ab) : C'est un langage qui implante les points de vue en utilisant l'héritage multiple. L'approche est fondée sur la sous-classification. Une classe est définie pour détenir l'identité d'un objet donné, puis des sous-classes sont définies pour chaque point de vue à représenter. Une sous-classe ne définit pas de nouvel objet. Elle peut être dérivée en sous-classe pour représenter plus finement son point de vue.

L'objectif principal de ROME est d'offrir une représentation multiple et évolutive d'objets. Un objet ROME a trois caractéristiques : des attributs, des méthodes et des sélecteurs. Les attributs et méthodes définissent son état interne, tandis que les sélecteurs décrivent son interface pour l'envoi de messages, ce qui permet à un objet d'être représentant de plusieurs classes. Cependant, vu son principe très dynamique, le langage ROME ne permet pas de fournir un moyen de contrôle de type statique. L'extension FROME tente de résoudre ce problème en intégrant le contrôle de type aux mécanismes proposés dans ROME (Carré et al. 1991, Dekker et al. 1992, Dekker 1992, Dekker 1994).

- LOOPS (Stefik et al. 1986, Bobrow et al. 1983) : Ce langage utilise la relation d'agrégation pour implanter les points de vue. Ceci est réalisé par des objets composites en permettant à chaque point de vue de n'utiliser qu'une partie de ses composants. Le langage LOOPS possède un typage statique et des mécanismes de gestion de la cohérence inter-vues. Cependant, il ne permet pas l'évolution des points de vues.

- SYSTALK (Wolinski et al. 1991, Perrot et al. 1992) : C'est un exemple de langage qui implante les points de vue en utilisant la délégation. Il est fondé sur SMALLTALK, mais il ne permet ni l'évolution des points de vue ni le typage statique.

I.3.2. Vues et Génie Logiciel

La modélisation du processus de développement est un domaine de recherche dans lequel la notion de point de vue a eu beaucoup d'intérêt. Nous nous intéressons ici aux travaux de Finkelstein et Charrel qui nous paraissent représentatifs du domaine.

L'approche de Finkelstein et al. (Finkelstein et al. 1990, Finkelstein et al. 1993, Nuseibeh et al. 1996) consiste à expliciter les expertises concernant le processus de développement d'une part et le métier du système d'autre part. Pour ces auteurs un point de vue est un "*objet faiblement couplé, géré localement, encapsulant les connaissances sur les modes de représentation, sur le processus de développement et sur les spécifications partielles d'un système et de son domaine*"[3]. Un point de vue est divisé en cinq parties (*slot*) : un *style* ou le formalisme de représentation ; un *plan* qui décrit les actions, le processus et les stratégies de développement ; le *domaine* qui détermine un champ d'intérêt particulier par rapport au système global ; la *spécification* qui est la description d'un domaine par un style, et *l'historique* qui garde une trace des actions exécutées dans le processus du développement.

---

[3] "A ViewPoint is a loosely coupled, locally managed object which encapsulates partial knowledge about the application domain, specified in a particular, suitable formal representation, and partial knowledge of the process of software development" (Finkelstein et al., 1990).

Dans cette approche une méthode de développement est une collection de *templates* (combinaison des *slots style* et *plan).* Une méthode de développement par point de vue d'un système consiste alors en l'instanciation de ces templates pour les différents domaines du système et en la gestion des différents types de corrélation inter-points de vues.

Cette approche permet donc la réutilisation des templates pour la représentation de différents systèmes. Par ailleurs, elle permet de mettre l'accent sur les différentes expertises concernant le métier et le processus de développement. La difficulté majeure de cette approche réside dans l'établissement des relations entre les points de vues.

Le travail de Charrel et al. (Charrel et al. 1993ab, Charrel et al. 1994) se situe aussi dans le contexte de développement de systèmes complexes. L'approche s'inscrit dans une démarche sémiotique qui a conduit à définir un point de vue comme "*le lieu d'interprétation d'un objet par un acteur : il est défini par l'objet sur lequel s'exerce l'interprétation, l'acteur qui exerce l'interprétation, l'expression et le contenu de l'interprétation de l'objet par l'acteur et le contexte dans lequel cette interprétation s'établit.*" (Charrel, 2000). A l'instar de Finkelstein, le paradigme de point de vue proposé par Charrel et al. se base sur les corrélations entre points de vue. Il se focalise sur la gestion des connaissances du processus de développement et la gestion de fragments de spécifications incohérents (Perrussel, 1998).

I.3.3. Vues dans les langages de programmation

La notion de vue/point de vue a été intégrée dans plusieurs langages à objets. Ces derniers utilisent les relations classiques du paradigme objet (héritage, délégation et composition) pour implanter cette notion. De même, certains travaux ont tenté d'intégrer les points de vues dans les langages à prototypes. Dans le reste de cette section, nous passons en revue des travaux de recherche concernant l'utilisation des vues/points de vue ou de concepts voisins dans les langages de programmation, à savoir : programmation par sujet (Harrison et al., 1993), programmation par aspect (Kiczales et al., 1997), programmation par vues (Mili et al. 1997, Mili et al. 2000, Mili et al. 2002), Programmation par objets structurés en contextes CROME (Vanwormhoudt 1999, Debrauwer 1998) et la programmation par points de vue en langages à prototypes (Bardou, 1998).

I.3.3.1. Programmation par sujets

La subjectivité comme concept de programmation a été introduite par (Harrison et al., 1993). La subjectivité permet d'identifier un ensemble de spécifications et de comportements reflétant la perception du monde réel correspondant à une vision générique d'un acteur. Cette approche a été généralisée par ses auteurs pour exprimer la séparation multidimensionnelle des préoccupations. Cette séparation permet de couvrir les différents types de préoccupations (métier, technologiques, règles de gestion, etc.). Le *concept de sujet* a été donc substitué par le concept d'*hyperslice* qui permet de décrire une préoccupation d'un système de manière multidimensionnelle et parallèle.

Dans cette section, nous abordons dans un premier temps les concepts de la programmation par sujet, puis nous présentons la séparation multidimensionnelle des préoccupations.

I.3.3.1.1. Concepts de la programmation par sujets

Le terme de sujet a été défini par Harisson et al. comme "une collection de spécifications d'états et de comportements qui reflètent une *gestalt* particulière, une perception du monde dans son ensemble,

tel qu'il est vu par une application particulière ou un outil particulier"[4]. Un sujet est une abstraction d'un processus générique ou d'une perception du monde. La figure 10 montre un exemple de perception d'objet selon trois sujets. Cet exemple montre qu'un administrateur a ses propres caractéristiques pour un *cours* : un prix et des méthodes de gestion. Ces caractéristiques peuvent s'appliquer à n'importe quel objet à vendre : produit, immobilier, etc. Ces caractéristiques sont dites alors extrinsèques à l'objet "Cours". Elles font partie de la vue subjective de l'administrateur. Un sujet ne correspond pas à une classe mais c'est une spécification de hiérarchie de classes. Cette hiérarchie dénote l'ensemble de descriptions d'objet pour une vision particulière. Cette spécification ne décrit pas de structure pour des instances objet mais c'est une description schématique qui peut s'appliquer à un domaine particulier d'objet. Ce sont les instances d'un sujet, appelés *activations de sujet*, qui contiennent effectivement les données d'un système. Un sujet peut être activé pour plusieurs domaines. De son coté, un objet peut activer plusieurs sujets. Le lien entre les différentes activations est réalisé à travers la notion d'identité d'objet. Un objet est identifié de manière unique dans toutes les activations. L'univers complet d'un système est l'union des différents sujets actifs. La composition (Ossher et al., 1995) de plusieurs sujets en une seule hiérarchie consiste en :

- la composition des interfaces émanant des différents sujets ;
- la composition de l'implémentation des définitions des différentes méthodes définies dans les différents sujets. L'homonymie est gérée par un langage de composition défini dans (Ossher et al., 1995). Les règles de composition (Ossher et al., 1995) utilisent des abstractions de classes nommées "subject labels".

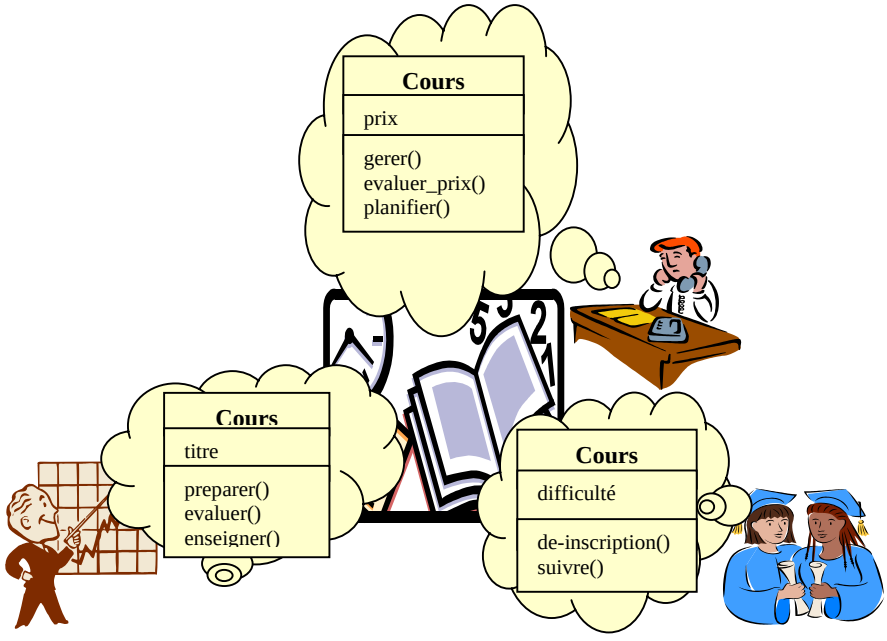

Figure 10 – *Plusieurs vues subjectives sur l'objet "Cours"*

La décomposition d'un système en fonction d'un ensemble de perspectives d'acteurs présente des difficultés dans le choix des limites entre les différents sujets. Pour supporter la décomposition des systèmes selon des préoccupations chevauchantes, les auteurs de l'approche par sujet ont introduit un

[4] "We use the term subject to mean a collection of state and behavior specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool" (Harisson et al., 1993)

nouveau type de décomposition dit multidimensionnel. Cette décomposition permet d'exprimer les différentes préoccupations d'un système durant les différentes phases du cycle de vie d'un système.

I.3.3.1.2. MDSOC : MultiDimensional Separation Of Concern

La séparation multidimensionnelle des préoccupations (Tarr et al., 1999) consiste à décomposer le développement de logiciel selon plusieurs niveaux (fonctionnel, métier, règle de gestion, technologique, etc.). Ces niveaux sont appelés des dimensions. La structure modulaire qui reflète cette décomposition est appelée un **hyperslice**. Un "hyperslice" ne contient que les unités pertinentes à une préoccupation particulière. La figure 11 ci-dessous illustre l'exemple de deux "hyperslices".

L'hyperslice « Formation » contient les unités et les propriétés concernant la préoccupation de la formation. Cet hyperslice contient les entités, les propriétés et les méthodes pour la préoccupation de gestion des formations. L'hyperslice « Inscription » contient les unités et les propriétés concernant la préoccupation d'inscription. Cette décomposition est une décomposition selon une dimension fonctionnelle. Le système peut être décomposé selon d'autres dimensions. Par exemple une décomposition selon la dimension technologique peut donner lieu aux hyperslices « Persistance », « Sécurité » , etc.

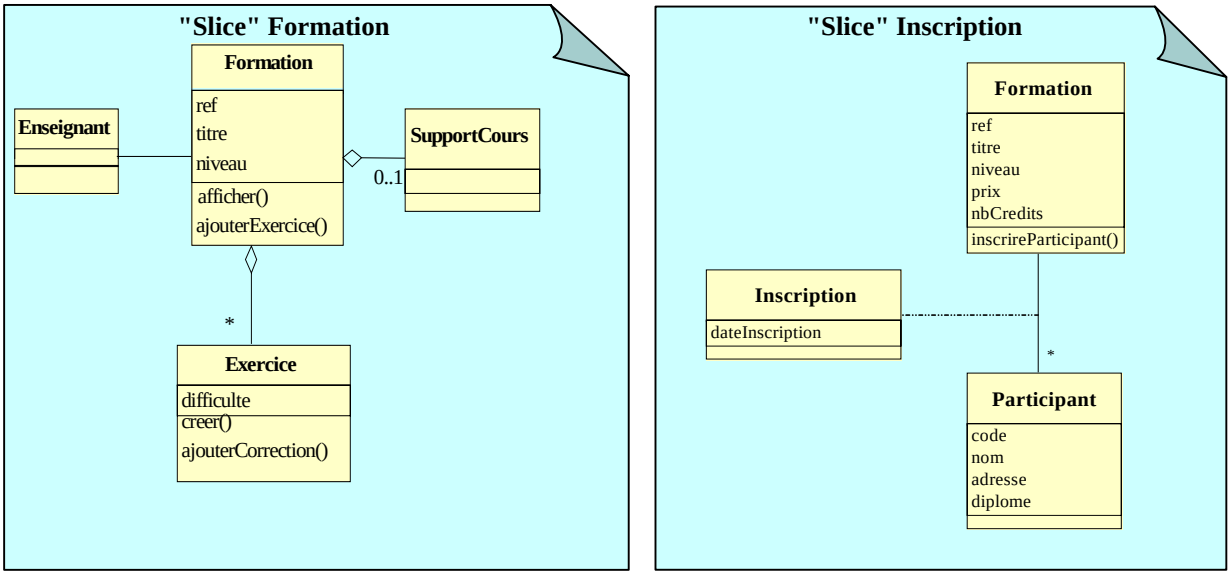

Figure 11 – *Exemple d'hyperslices*

Un système est une collection d'hyperslices composés dans un **hypermodule** selon des règles de composition. Pour composer un ensemble d'hyperslices, le développeur doit effectuer un ensemble de déclarations dans un hypermodule en indiquant l'ensemble des hyperslices à composer et les règles de composition. L'outil de composition est appelé Hyper/J (Ossher et al., 2001). Hyper/J supporte MDSOC pour le développement JAVA . Il opère sur les 'class file' pour intégrer les différents hyperslices d'un hypermodule.

La figure 12 montre un exemple de déclaration Hyper/J. Cette déclaration montre que les hyperslices à composer sont « Formation » et « Inscription » en utilisant la règle de composition « Merge by name ».

```
hypermodule InscriptionPlusFormation
hyperslices:
Feature.Inscription, Feature.Formation;
relationships:
mergeByName;
…
end hypermodule
```

Figure 12 – *Un exemple composition par Hyper/J*

I.3.3.1.3. Discussion

L'approche par sujet permet une flexibilité d'adaptation d'objets. Elle met l'accent sur le sujet comme unité principale d'abstraction. Cette approche est caractérisée par :

Absence de classe globale : Dans la programmation par sujet, il n'y a pas de classe globale décrivant l'univers d'un objet mais chaque sujet actif donne lieu à une classe de description de propriétés et des implémentations de méthodes propres au sujet.

Activation de points de vue : Dans un système distribué, plusieurs sujets peuvent être activés pour un objet. Cette activation multiple de sujets est appelée univers d'activation. L'univers d'activations de sujets est un couple (OID, SA) où OID est l'ensemble des identificateurs d'objet et SA l'ensemble des activations de sujets. Chaque activation spécifie l'état d'un OID pour une activation de sujet. Les différentes activations partagent le même ensemble OID.

Cohérence de données : L'approche par sujet propose des règles de composition pour propager des données suite à une invocation de méthode. Cette propagation permet de gérer la cohérence dans un univers de sujets actifs.

Invocation de méthodes : Dans la programmation par sujet, l'invocation de méthode se fait à l'intérieur d'un univers de sujets. Plusieurs exécutions de méthodes sont alors possibles. La bonne exécution est déterminée par les règles de composition d'implémentation.

Classification dynamique : L'approche par sujet ne permet pas la classification dynamique d'objet car l'activation de sujet est déterminée à la compilation.

I.3.3.2. Programmation par aspects

I.3.3.2.1. Présentation de l'approche

La programmation par aspect (Kiczales et al., 1997) permet d'encapsuler les exigences non fonctionnelles d'un système sous forme d'aspects. Les exigences non fonctionnelles correspondent aux qualités ou contraintes imposées pour mieux satisfaire les besoins fonctionnels d'un système. Il s'agit d'aspects de performance, de sécurité, de persistance, de journalisation, d'accès aux ressources, d'optimisation, d'authentification, etc. L'implantation de telles exigences se trouve souvent enchevêtrée avec les différents modules fonctionnels d'un système. Ce recoupement de préoccupation est désigné par le terme anglais "*crosscutting concern*". Le principe de la programmation orientée aspect est de coder chaque problématique séparément et de définir leurs règles d'intégration pour les combiner en vue de former un système final. Le tisseur *(Weaver)* est l'infrastructure qui permet de greffer le code

des aspects dans le code des méthodes des classes. La figure 13 ci-après montre un exemple de recoupement de préoccupation.

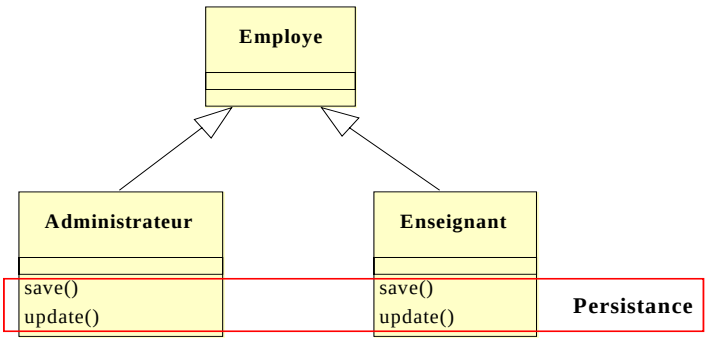

Figure 13 – *Exemple de recoupement de préoccupations*

Sur le diagramme de classes UML de la figure 13, on suppose qu'on souhaite enregistrer les informations concernant des employés suite à chaque mise à jour. La persistance est une préoccupation qui concerne les deux classes « Administrateur » et « Enseignant ». On parle alors de recoupement des préoccupations. En général, on dit que des préoccupations se recoupent si plusieurs méthodes sont concernées par ces aspects (Elrad et al., 2001). La programmation orientée aspect (Kiczales et al., 2001) permet d'encapsuler ces préoccupations dans des modules réutilisables.

(Kiczales et al., 1997) distingue deux types de modules, composants et aspects, en donnant les définitions suivantes :

- Un **composant** est un élément qui peut être clairement encapsulé dans un objet, ou dans un module. Les composants sont les unités fonctionnelles d'un système.

- Un **aspect** n'est pas une unité de décomposition de système mais une propriété qui affecte la sémantique des composants d'un système. C'est une fonctionnalité qui ne peut pas être encapsulée à l'aide des moyens de structuration conventionnels. Les aspects correspondent aux exigences non-fonctionnelles d'un système.

Le but de la POA est de séparer la programmation des composants de celles des aspects et de disposer de moyens de tissage (*weaving*) pour composer le système final sur la base des différents modules et des règles d'intégration. La figure 14 donne un exemple de tissage de code. Elle montre comment le code de la sauvegarde des nouvelles données à chaque changement est greffé dans le code des méthodes de la classe « *Administrateur* ».

I.3.3.2.2. AspectJ

AspectJ (Kiczales et al., 2001) est un langage populaire qui implémente les spécifications de la programmation orientée aspect. AspectJ est une extension de Java. Les composants sont écrits en Java pur. Grâce aux possibilités offertes par la syntaxe d'AspectJ, il est possible de définir des aspects accompagnés de leurs règles d'intégration. Ces règles s'expriment sous forme de points de jointure, "*pointcuts*" et "*advice*". Nous pouvons résumer les différents concepts introduits par AspectJ comme suit :

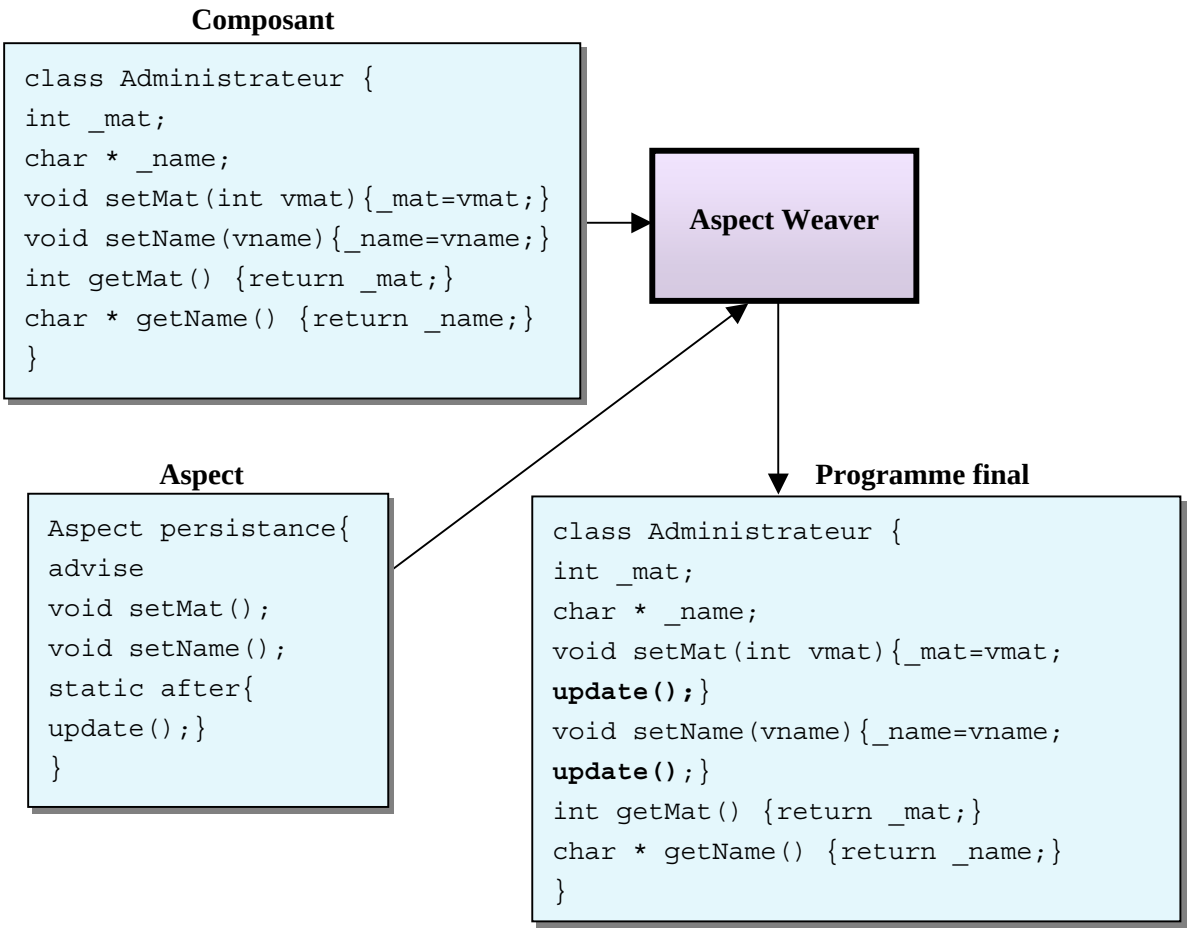

Figure 14 – *Intégration d'aspect dans un composant*

Aspect : Un aspect est similaire à une classe Java, dans le sens où il contient des champs et des méthodes et peut même étendre d'autres classes ou d'autres aspects.

Point de Jointure : Les points de jointure désignent des points précis dans l'exécution d'un programme. AspectJ permet de définir les points de jointure suivants :

- un appel de méthode ou de constructeur (dans le contexte de l'appelant),
- l'exécution d'une méthode ou d'un constructeur (dans le contexte de l'appelé),
- l'accès en lecture ou écriture d'un champ,
- l'exécution d'un bloc "*catch*" qui traite une exception Java,
- l'initialisation d'un objet ou d'une classe (c'est-à-dire des membres statiques d'une classe).

PointCut : Les *pointcuts* correspondent à la définition syntaxique d'un ensemble de points de jointure, plus, éventuellement, certaines valeurs dans le contexte d'exécution de ces points de jointure.

Advice : Un *advice* est un mécanisme (similaire à une méthode) utilisé pour déclarer l'exécution de code aux points de jointure d'un "pointcut". Il y a trois types *d'advice* : les "*before advices*", les "*around advices*" et les "*after advices*". Comme leurs noms l'indiquent, les "*before advices*" s'exécutent avant que les points de jointures ne soient exécutés ; les "*around advices*" permettent d'exécuter du code avant et après les points de jointure ; les "*after advices*" s'exécutent uniquement après l'exécution des points de jointure.

La programmation avec AspectJ utilise en même temps les objets et les aspects pour séparer les préoccupations. Chaque aspect définit une fonctionnalité qui transcende le modèle objet. Un aspect peut déclarer des attributs et des méthodes. Il peut aussi étendre un autre aspect en concrétisant des

comportements abstraits. Le "*crosscutting*" permet d'introduire de nouvelles méthodes dans des classes existantes. Il peut aussi changer la hiérarchie des classes. Les aspects peuvent aussi affecter la structure dynamique d'un programme en changeant la manière d'exécuter ce programme en interceptant les "*joinpoints*" avec des évènements "*before*", "*after*" et "*around*".

I.3.3.2.3. Discussion

La programmation par aspect permet l'écriture de code flexible et adaptable à des besoins donnés. Elle permet une adaptation à des exigences du système. On peut parler de point de vue système. Cependant, la flexibilité offerte par les approches de la programmation par aspect est souvent statique. En effet, dans la POA, l'intégration d'un aspect dans un composant est une tâche qui a lieu à la compilation sans pouvoir évoluer ou changer dynamiquement. Mais des implémentations comme celles proposées par (Pryor et al., 1999) et (Joshi et al., 2002) essaient de fournir des tissages dynamiques d'aspects en se basant sur la réflexion (Maes, 1987).

L'application de la programmation par aspect pour la gestion de la distribution et la gestion des vues d'un système s'avère prometteuse. En effet, la gestion de vues et la distribution d'objet multivues sont des préoccupations orthogonales aux systèmes multivues distribués.

I.3.3.3. Programmation par vues

Dans un système d'information, les mêmes entités jouent différents rôles fonctionnels. Ces rôles sont destinés à différents usagers, et leur implantation implique des compétences différentes. Le travail de Mili et al. (Mili et al. 1997, Mili et al. 2000, Mili et al. 2001, Mili et al. 2002) traite le problème du développement, de la réutilisation, de la maintenance et du déploiement séparé de ces rôles fonctionnels. Ce travail est centré autour d'un ensemble d'outils logiciels qui implantent la programmation par vues, où une vue est une construction logicielle réalisant un rôle fonctionnel.

I.3.3.3.1. Principes de la programmation par vues

Chaque objet du domaine d'application supporte un ensemble de fonctionnalités de base offertes, directement ou indirectement, à tous les utilisateurs (de l'objet), et d'autres fonctionnalités ou vues, spécifiques à certains utilisateurs. L'approche de programmation par vues proposée par Mili et al. consiste à représenter un objet par un « objet de base », et un ensemble variable de vues représentant des facettes fonctionnelles, chacune avec ses propres données et ses propres comportements, mais accédant aux données et services de l'objet de base. Durant sa vie, chaque objet supporte un certain comportement de base, et un ensemble variable de vues (comportements), prises dans l'ensemble des vues applicables à la classe, actives simultanément, que l'on peut ajouter ou retirer sur demande. L'ensemble des vues attachées à un objet à un instant donné détermine son comportement. Concrètement, le modèle proposé est implémenté par un ensemble d'objets (objets vues qui pointent vers un objet base). Concernant le partage des variables, il est fait par un mécanisme de délégation, tandis que le partage de comportement est réalisé par un mécanisme de redirection de message. L'invocation d'un comportement supporté par plusieurs vues est traité en adoptant l'approche de Harrison et Ossher pour la composition de sujets qui consiste à combiner les différentes implantations (Harrison et al. 1993, Ossher et al. 1995). Cette solution repose sur la notion de « vue universelle de composition » qui est générée automatiquement à partir de la liste des vues potentielles, et qui engendre des germes de méthodes pour les comportements supportés par plusieurs vues.

La figure 15 illustre l'implémentation (basée sur l'agrégation) adoptée par Mili et al. pour les objets avec vues. L'objet représenté dans cette figure consiste en un objet de base et trois vues. L'appel de la fonction *f()* sur la *vue1*, est retransmis à l'objet de base et sera exécuté dans le contexte de cet objet de base. Il en est de même pour les références aux variables partagées (« a » pour *vue2*, et « b » pour *vue3*). Concernant la fonction *k()*, deux vues (1 et 2) en offrent des implantations ; dans ce cas des règles de composition permettent de spécifier la façon de combiner les deux implantations (Ossher et al., 1995).

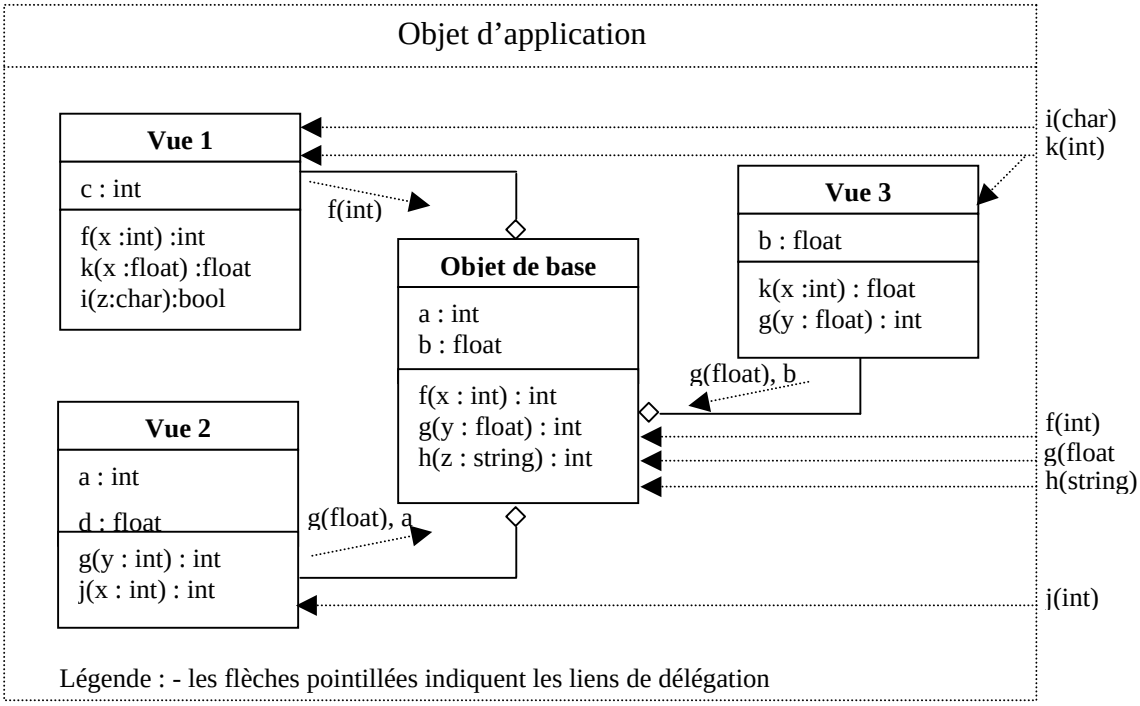

Figure 15 – *Un modèle d'objets avec vues (tiré de (Mili et al., 2001))*

I.3.3.3.2. Vues et points de vues

Dans les systèmes d'information d'entreprises, les rôles fonctionnels joués par les objets du domaine correspondent souvent à des processus d'affaires génériques qui ne dépendent pas du domaine d'affaire. La programmation par vues proposée par Mili et al. représente les rôles fonctionnels par des vues. La logique inhérente à ces vues peut être perçue comme une instanciation d'une logique plus générique, qui correspond à un processus d'affaires générique. Par exemple, pour le service de la comptabilité, un avion, une voiture, un ordinateur, sont toutes des pièces d'équipement ayant un prix d'acquisition, qui peut être amorti sur une période de temps. Ce « comportement comptable » pourrait être codé de façon générique et instancié pour différents objets métiers. De là, un point de vue est défini comme une sorte de classe générique dont le type paramètre est l'objet du domaine auquel le comportement générique pourrait s'attacher. Par exemple, il est possible de définir le point de vue *CapitalAmortissable (Materiel)*, Où *Materiel* est un paramètre (type) qui représente le fait d'être un matériel ayant un prix, une date d'acquisition, etc. Le point de vue *CapitalAmortissable* spécifie les primitives additionnelles concernant l'amortissement de matériel. Cette notion de point de vue permet en plus la réutilisation de vues, elle offre un moyen pour découpler le développement des vues du développement des objets de base.

I.3.3.3.3. Implémentation de la programmation par vues

Un point de vue est considéré comme un type paramétré par une théorie. Cette dernière permet de spécifier le type des objets auxquels le point de vue peut être appliqué. La figure 16 montre un exemple de définition d'un point de vue. La théorie est spécifiée dans la clause *REQUIRES* (*Matériel* dans l'exemple). La clause *EXTENDS* est utilisée pour spécifier des blocs de code, exécutés par la vue, avant (*before*) et après (*after*) des méthodes de l'objet de base.

```
VIEWPOINT CapitalAmortissable
      REQUIRES Matériel
            EXTENDS
            void setDateAcquisition(Date d)
                  after
                  { setPériodeAmortissement(Year(Date::today() - d) ; }
      PROVIDES
            variables
                  Year  _périodeAmortissement;
                  TypeMonnaie  _valeurRésiduelle;
            operations
                  TypeMonnaie getValeurResiduelle ()    {
                        return   _valeurRésiduelle ; }
                  void setPériodeAmortissement(Year y) {
                        _périodeAmortissement = y ; }
                  Date getAge () {
                        return Date::today()-getDateAcquisition() ; }
END  VIEWPOIINT
```

Figure 16 – *Exemple de point de vue (Mili et al., 2001)*

En C++, un point de vue est instancié selon la syntaxe suivante :

```
Defview FCAmion as CapitalAmortissable [Camion,
      (setDateAcquisition(Date), setDateAchat(date))]
```

Cet exemple d'instanciation de point de vue applique le point de vue *CapitalAmortissable* à la classe de base *Camion*. Le résultat est la classe (vue) *FCamion*. La méthode *setDateAcquisition()* est une opération exigée par la théorie *Matériel*. L'absence de cette opération dans la classe *Camion* est palliée par la substitution « *setDateAcquisition(Date) -> setDateAchat(date)* » qui spécifie que l'opération *setDateAchat(date)* joue le même rôle que *setDateAcquisition(Date)*.

La figure 17 illustre la classe *Camion* et la classe de la vue *FCamion* générée. La classe *Camion* doit hériter de la classe *Viewable* qui possède des primitives de gestion des vues. Les méthodes et attributs déclarés dans la clause *PROVIDES* du point de vue *CapitalAmortissable* deviennent des méthodes et attributs de la classe *FCamion*. Si une méthode de *FCamion* fait référence à un attribut de l'objet de base, une transformation est faite de façon à tenir compte de la délégation. Par exemple, la méthode « *Date getAge()* » qui fait partie de la clause *PROVIDES* est transformée dans la classe *FCamion* comme suit :

```
Date FCamion::getAge() {
          Return Date ::today()- camion->getDateAchat() ; }
```

class Camion : public Viewable, … {
public :
…
TypeNSerie getNumeroSerie();
TypeMarque getMarque();
Year getAnnéeModele();
Date getDateAchat();
float getCharge();
protected :
…
private :
Type _id;
… };
	class FCamion {
	public :
	FCamion(TypeId unId);
	…
	float getValeurResiduelle();
	float getPeriodeAmortissement();
	void setValeurResiduelle(float);
	void setPeriodeAmortissement(float) ;
	Date getAge() ;
	private :
	Camion *_camion ;
	Year _periodeAmortissement ;
	TypeMonnaie valeurResiduelle ;
	…} ;

Figure 17 – *Les classes Camion et FCamion*

I.3.3.3.4. Gestion des vues et aiguillage d'appels

Supposons qu'en plus de la vue *FCamion*, il y ait deux autres vues *OCamion* et *ECamion*, reflétant les rôles *opération* (cédule, affectation de charges, etc.) et *entretien* (cédule, coût, temps d'arrêt, etc.). L'exemple suivant illustre un extrait d'un programme C++ utilisant les trois vues *FCamion*, *OCamion* et *ECamion*.

```
// Inclusion des définitions de la classe de base et des vues
#include <camion.h>
#include<fcamion.h>
#include<ocamion.h>
#include<ecamion.h>

// création de l'objet de base
Camion*  monCamion = new Camion(id);
...
// attachement de la vue FCamion à la base
cout  <<"Num. Série : "<<monCamion->getNumeroSerie();
monCamion->attach("FCamion") ;
// appel de la fonction setValeurResiduelle() sur monCamion
// la fonction setValeurResiduelle() est définie dans FCamion
monCamion->setValeurResiduelle(15000.0) ;
// transformation de cette ligne par le préprocesseur :
 ==> monCamion->getView("FCamion")->setValeurResiduelle(15000.0) ;
// appel de la fonction ceduleChargeEntre() sur monCamion
// la fonction ceduleChargeEntre() est définie sur la vue OCamion
monCamion->ceduleChargeEntre(charge,date1,date2) ;
```

```
//transformation de cette ligne par le préprocesseur :
 ==>
monCamion->getView("OCamion")>ceduleChargeEntre(charge,date1,date2);
 ==> Erreur d'exécution :la vue OCamion n'est pas attachée à la base
// attachement de la vue OCamion à la base
monCamion->attach(”OCamion”) ;
monCamion->ceduleChargeEntre(charge,date1,date2) ;
// inhiber le comportement de la vue FCamion sans la détruire (pour
// préserver ses données pour une prochaine activation)
monCamion->deactivate(”FCamion”) ;
```

Dans le cas où un comportement est supporté par plusieurs vues qui sont attachées à la base, Mili et al. ont adopté une approche qui consiste à utiliser des règles de composition par défaut, que les développeurs pourront redéfinir (override) localement (Ossher et al., 1995), selon la sémantique du domaine d'application.

I.3.3.3.5. Discussion

L'approche de Mili et al. exploite le constat que les rôles fonctionnels joués par les objets du domaine correspondent souvent à des processus génériques qui ne dépendent pas du domaine d'affaire. Ces rôles fonctionnels sont représentés par des vues. Cette approche s'intéresse au problème du développement, de la réutilisation, de l'entretien et du déploiement de ces vues. Cependant, elle n'offre pas de démarche pour élaborer une analyse/conception par vues. Cette approche rejoint celle de VBOOM en ce qui concerne la possibilité d'activer plusieurs vues simultanément sur un objet donné.

Sur le plan implémentation, l'approche de Mili et al. souffre d'un inconvénient lié à l'aiguillage des appels vers les objets supportant les méthodes demandées ; en effet, cet aiguillage est réalisé statiquement à l'aide d'un préprocesseur.

I.3.3.4. Un cadre de programmation par objets structurés en contexte : CROME

Les systèmes logiciels orientés objet sont en général organisés sous forme de référentiels d'objets et de fonctions impliquant ces objets. Le but du travail de Vanwormhoudt (Vanwormhoudt, 1999) est de corriger l'inconvénient de cette double structuration objet/fonction en proposant une solution par contexte. La structuration par contexte est issue des notions de point de vues de ROME (Carré, 1989). La solution CROME (Contextes en ROME) consiste à identifier les objets de base d'un système et les contextes d'intervention de ces objets qui se traduisent par des enrichissements spécifiques à chaque contexte. Au sein d'un contexte, les objets se décrivent comme dans un modèle classique à objet tout en permettant des moyens d'articulation entre les fonctions du système.

I.3.3.4.1. Structuration par plans

L'approche CROME est basée sur une structuration en plans. En fait, l'application de CROME conduit à identifier un référentiel d'objets sous forme de plans de base. Ce référentiel est étendu par les différentes fonctions sous forme de plans fonctionnels. Vanwormhoudt (Vanwormhoudt, 1999) distingue ainsi deux sortes de plans :

- **Plan de base :** détermine l'identité des objets du référentiel. Les descriptions contenues dans ce plan sont partagées par toutes les fonctions. Les objets y sont décrits par des relations d'héritage et des relations traditionnelles entre objets.
- **Plan fonctionnel :** décrit pour les objets de base les spécificités fonctionnelles par rapport à une fonction donnée.

La figure 18 ci-dessous présente un exemple de description d'un circuit. Le plan de base comprend la description des composants internes de sa structure. Cette structure est partagée par les différentes fonctions de l'application. Le rectangle au centre contient la description de base du circuit alors que les rectangles qui l'entourent sont des descriptions fonctionnelles qui correspondent, respectivement, aux fonctions de documentation, d'optimisation, de normalisation, de simulation et d'édition.

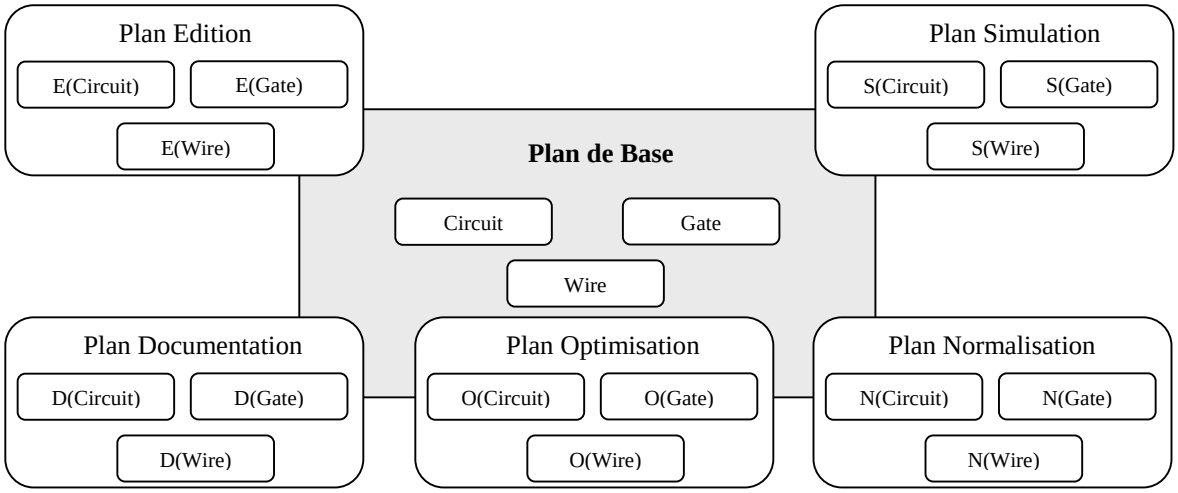

Figure 18 – *Plan de base et plans fonctionnels pour un système électronique (tiré de (Vanwormhoudt, 1999))*

La description de base décrit les attributs des objets partagés par toutes les fonctions (Circuit, Wire et Gate). Cette description peut être plus ou moins riche selon le niveau de détail souhaité. Les plans fonctionnels sont des enrichissements des classes du plan de base. Ils n'introduisent pas de nouveaux objets, mais étendent la base pour répondre aux spécificités des différents contextes du système. L'enrichissement d'une classe de base pour une fonction donnée est appelé **Partie fonctionnelle**. Cette partie étend la description de base de la classe en ajoutant des attributs et des méthodes n'ayant de sens que pour ce plan fonctionnel. Cette partie n'est pas instanciable, elle préserve l'identité de l'objet de base.

La **structure d'une partie fonctionnelle** se résume en un ensemble d'attributs et de méthodes privés et publiques. Ces attributs viennent compléter la classe pour cette fonction et ne sont, par conséquent, accessibles que par les méthodes de cette partie fonctionnelle. La figure 19 ci-dessous illustre la définition de la partie fonctionnelle associée à la classe *AndGate* pour la fonction d'optimisation.

Un plan fonctionnel préserve les relations d'héritage issues du plan de base. Une partie fonctionnelle hérite systématiquement de la classe de base. Elle peut redéfinir les méthodes de la classe de base pour les adapter ou les enrichir de façon spécifique à une fonction. Une partie fonctionnelle est identifiée par le nom du plan fonctionnel et la classe dont elle dérive. Au sein d'un plan fonctionnel, la partie fonctionnelle enrichissant une classe est héritée localement par ses classes

descendantes. Ceci donne lieu à un héritage contextualisé entre les classes au travers de leurs parties fonctionnelles. De plus, une classe qui n'est pas explicitement enrichie par une partie fonctionnelle dans un plan, l'est implicitement en héritant localement des parties fonctionnelles associées à ses classes parentes.

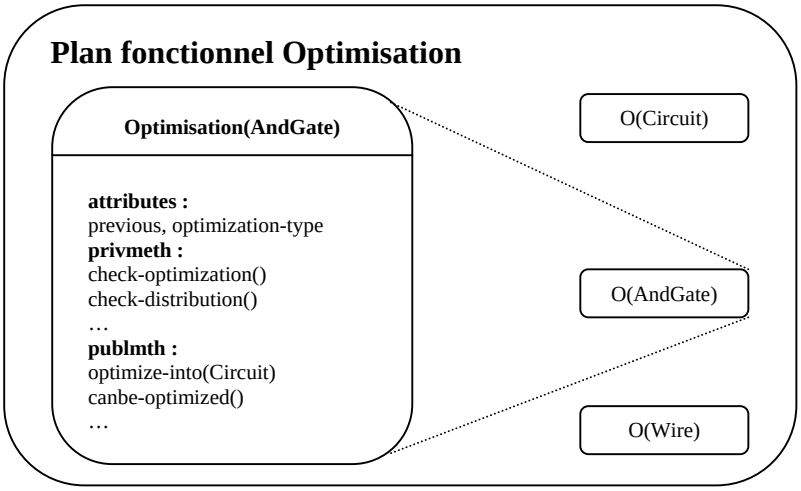

Figure 19 – *Partie fonctionnelle de la classe AndGate pour la fonction d'optimisation*

La description complète d'un objet est déterminée par sa description de base et l'ensemble des parties fonctionnelles associées.

I.3.3.4.2. Envoi de messages entre objets dans un même contexte

Un contexte est l'association d'un plan fonctionnel et du plan de base. Un contexte est un vrai programme qui implique les différents objets pour l'accomplissement d'une fonction donnée. Par conséquent l'envoi de messages entre objets au sein d'un même contexte est basé sur leurs descriptions au sein de ce contexte. Un objet n'a donc qu'une vue partielle des autres objets selon le contexte. Le polymorphisme classique qu'induit l'envoi de messages est conservé au sein d'un contexte.

I.3.3.4.3. Appel de méthodes inter-contextes

Dans une partie fonctionnelle aucune restriction n'est imposée pour l'appel de méthodes d'autres parties fonctionnelles de la même classe. L'appel à une méthode d'une partie fonctionnelle appartenant à un autre contexte provoque le changement provisoire de contexte.

I.3.3.4.4. Discussion

L'approche proposée dans (Vanwormhoudt, 1999) est une approche modulaire. Le découpage par plans permet de regrouper les attributs et les méthodes liés à une même fonction. Par ailleurs, la modification d'un contexte ou l'ajout d'un nouveau contexte ne remet pas en cause les autres contextes. Ce découpage rejoint le découpage en modèles visuels proposé par VBOOM (Kriouile et al., 1995). (Vanwormhoudt, 1999) propose des règles de contextualisation pour la résolution des problèmes d'articulation entre contextes pour garder la cohérence au sein d'un objet alors que VBOOM propose la fusion des modèles visuels pour générer un modèle global accessible par les différents points de vue.

L'approche par contexte restreint le clientélisme entre objets à ceux qui appartiennent à un même contexte. Le changement de contexte d'un objet n'est permis que pour des appels de fonctions inter-contextes. Le changement dynamique de contexte est un atout principal pour les systèmes distribués. En effet, dans un système distribué où des objets serveurs participent à l'accomplissement de fonctions clientes, un objet serveur est candidat à changer son contexte. La contextualisation de graphe d'objet en CROME permet la factorisation des objets de base mais n'explicite pas le changement de comportement dynamique.

En résumé, CROME est un cadre de programmation qui améliore la modularité des systèmes par des contextes. Il propose une nouvelle forme de réutilisation autre que la composition ou l'héritage par l'élaboration de parties fonctionnelles. Cependant, il ne propose pas une démarche pour identifier les contextes d'un système ou circonscrire les parties fonctionnelles d'un objet. Par ailleurs le comportement dynamique et le problème d'identité unique d'un objet ne sont pas explicités dans CROME.

D'autres travaux ont repris l'approche CROME comme point de départ. Citons en particulier les travaux de Muller et al. (Muller et al., 2003) qui s'intéressent à la conception pour la réutilisation de composants de systèmes d'information adaptables dans leur dimension fonctionnelle. Ceci est réalisé en combinant le mécanisme de vues de CROME et la notion de composant. Cependant, Muller et al. ne proposent pas de démarche pour élaborer les composants.

I.3.3.5. Objets morcelés, délégation et points de vue

Apparus à la fin des années 1980, les langages à prototypes ont été proposés comme une alternative aux langages à classes, face auxquels ils proposent un modèle de programmation plus simple et moins limité. Les langages à prototypes proviennent d'horizons divers. Ils ont été inspirés par la théorie des prototypes développée en sciences cognitives et qui a notamment donné lieu à l'apparition des langages de frames en représentation des connaissances. Ils intègrent le clonage et la délégation comme mécanismes fondamentaux et présentent en ce sens certaines caractéristiques communes avec les langages d'acteurs. L'un des premiers arguments en faveur du modèle à prototypes est que celui-ci semble plus naturel, plus proche du processus de raisonnement humain que le modèle à classes. La programmation dans un langage à classes est essentiellement basée sur la définition des classes, soit sur une description abstraite de concepts. La représentation d'exemples et leur manipulation ne peut en aucun cas y prendre place avant la définition des classes. L'humain raisonne souvent en premier lieu sur des exemples d'un problème nouveau ou d'un concept avant de s'en construire une abstraction. Or il est difficile de déterminer exactement et a priori quelles caractéristiques sont essentielles dans la représentation d'un concept (Lieberman, 1986), et lesquelles ne relèvent que du détail ou de l'anecdote. Les systèmes à prototypes offrent la possibilité de représenter de prime abord un concept par un ou plusieurs exemples individuels, puis de généraliser la représentation obtenue en déterminant quels aspects peuvent varier d'un exemple à l'autre en raisonnant par analogie (Lieberman, 1986). Les prototypes permettent d'analyser les phénomènes en employant des métaphores et ne contraignent pas l'utilisateur à concevoir d'emblée de grandes hiérarchies conceptuelles (Taivalsaari, 1993).

Parmi les travaux qui ont tenté d'introduire les points de vue dans les langages à prototypes, nous citons, le travail décrit dans (Bardou, 1998) qui consiste en une étude des langages à prototypes, du mécanisme de délégation, et de son rapport à la notion de point de vue. L'étude d'une certaine forme du mécanisme de délégation mis en oeuvre dans les langages à prototypes a permis de mettre en évidence la possibilité d'utiliser ce mécanisme pour représenter une entité selon plusieurs points de

vue. Cette possibilité de représentation est directement liée au type de partage entraîné par la délégation et est de ce fait absente de la plupart des langages à classes, du moins ceux qui ne comportent aucun mécanisme spécifique à cette fin (Bardou, 1998). L'étude de cette possibilité de représentation, la recherche d'une solution permettant de mieux l'exploiter et l'analyse de la notion de point de vue associée constituent l'objectif principal de ce travail. (Bardou, 1998) propose les objets morcelés comme constructions particulières dans lesquelles l'usage de la délégation est discipliné et les messages sont évalués en prenant en compte une certaine notion de point de vue.

I.3.3.5.1. Description de l'approche

Bardou propose de rassembler toutes les représentations d'une entité sous forme d'objet unique. L'approche est basée sur le mécanisme de représentation éclatée d'objets dans lequel plusieurs objets sont utilisés pour la représentation d'une seule entité du domaine selon plusieurs points de vue, et confrontée au problème d'identification unique d'une entité du domaine.

La figure 20 représente un objet "Pierre" morcelé selon plusieurs points de vues. Seul l'ensemble de la représentation a le statut d'objet. Les points de vue sont dénotés par des morceaux de cet objet.

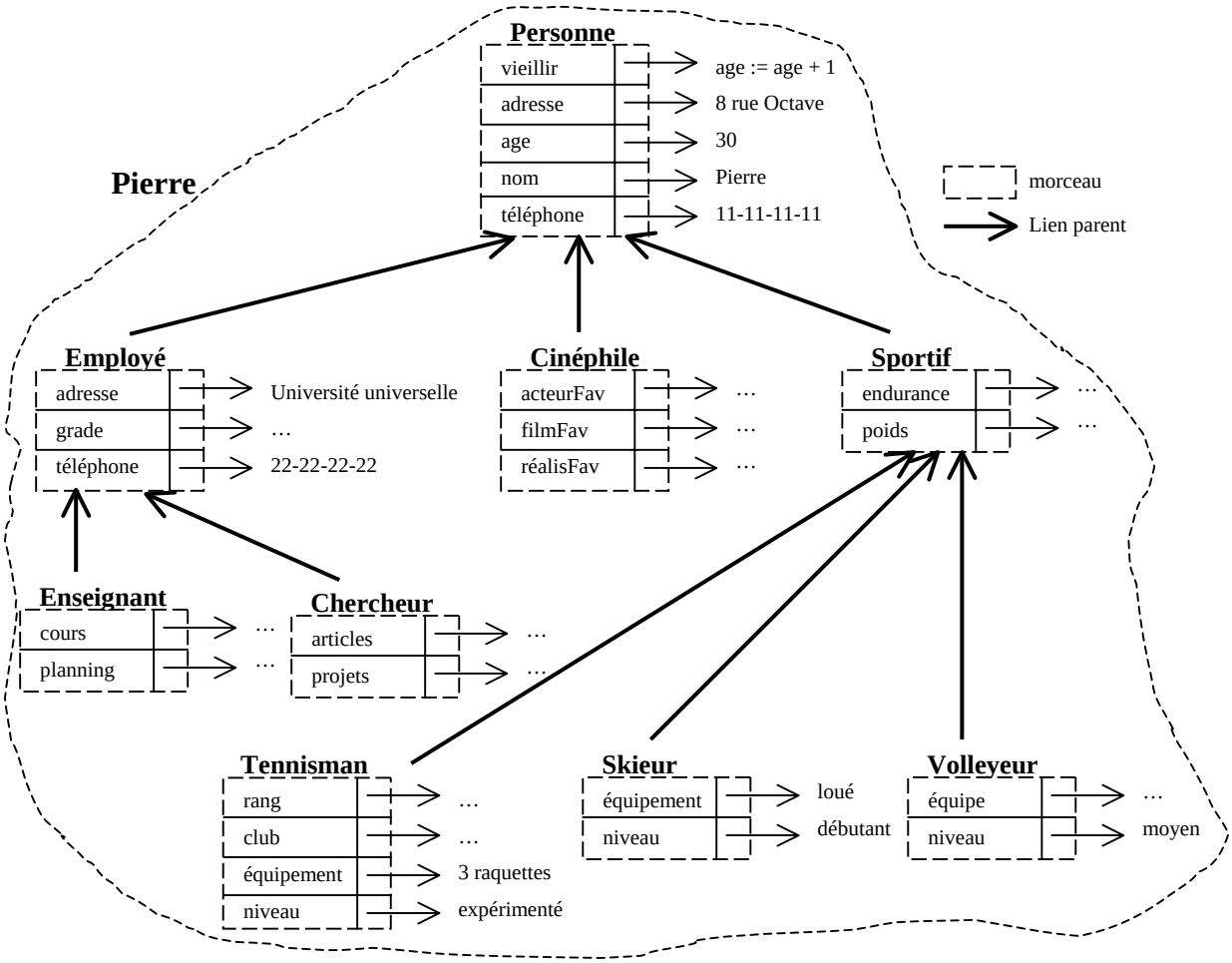

Figure 20 – *Représentation de l'entité Pierre par un objet morcelé*

Un objet morcelé est défini par une collection de *morceaux*. Ces morceaux n'ont pas le statut d'objet et ne sont pas instanciables. Cependant, chaque morceau est une unité d'abstraction qui détient

les propriétés spécifiques à un point de vue particulier. Les morceaux sont organisés au sein d'un objet morcelé selon une hiérarchie de délégation (avec partage de propriétés). Cette hiérarchie admet toujours une racine unique. Bardou justifie ceci par le fait que la racine peut être vide de propriétés. Toute propriété détenue par un morceau est également accessible depuis les morceaux descendants. Les descendants dénotent des points de vue spécialisés par rapport aux morceaux ascendants. Ainsi, le morceau racine dénote le point de vue le plus général, tandis que les morceaux feuilles dénotent les points de vues les plus spécifiques.

L'objet morcelé fournit une identité unique à toute la représentation d'une entité du domaine tout en permettant que celle-ci soit représentée sous plusieurs aspects. La notion de point de vue est intimement liée à l'activation des propriétés d'objet morcelé par envoi de message. Seul l'objet morcelé a le statut d'objet et peut recevoir des messages.

Bardou considère trois types de points de vue différents : points de vue simple, point de vue combiné et point de vue global. Un objet morcelé de *n* morceaux dénote 2^n-1 points de vue dont :

- n sont des points de vue simples, explicitement dénotés par un morceau,
- 2^n-n-1 sont des points de vue combinés, obtenus par combinaison des points de vue simples,
- 1 point de vue combiné particulier est appelé le point de vue global, il correspond à la combinaison de tous les points de vue simples de l'objet morcelé.

Par exemple, 511 points de vue sur Pierre peuvent être considérés lors de l'envoi de message à l'objet morcelé *Pierre*.

I.3.3.5.2. Manipulation des objets morcelés

- ***Nommage et accès :*** les objets morcelés peuvent être directement accessibles par les programmeurs, alors qu'un morceau d'objet ne peut être accessible qu'à travers l'objet morcelé englobant. Ce dernier peut interdire l'accès à l'un ou plusieurs de ses morceaux.
- ***Création :*** la création se fait par clonage ou création d'objet vide, puis ajout de morceaux.
- ***Clonage :*** il consiste en une copie profonde de la totalité de la hiérarchie de l'objet morcelé.
- ***Modification :*** on peut ajouter/supprimer des morceaux à/d'un objet morcelé.
- ***Envoi de messages :*** Seul l'objet morcelé, ayant le statut d'objet, peut recevoir des messages ; le nom du morceau dénotant le point de vue doit être spécifié par le programmeur. Le sélecteur de la propriété est alors recherché dans le morceau désigné, puis dans ses ascendants. Un point de vue combiné peut être spécifié comme collection de noms de morceaux et sa spécification revient donc à donner une collection de noms de morceaux lors de l'envoi du message. La recherche du sélecteur est alors limitée dans les morceaux spécifiés. L'envoi de message selon un point de vue global correspond à un accès à la représentation de l'entité dans sa globalité. Tous les morceaux sont alors impliqués suite à un envoi de message. Il est par exemple possible d'envoyer un message de sélecteur *poids* à l'objet morcelé *Pierre* en spécifiant comme morceau *Skieur*. Le sélecteur est d'abord recherché dans le morceau *Skieur*, puis dans son parent le morceau *Sportif* où il est trouvé. Il est aussi possible d'envoyer un message à *Pierre* en considérant *Pierre* à la fois en tant que *cinéphile* et *sportif*, c'est-à-dire en spécifiant les deux noms de morceaux *Cinéphile* et *Sportif* comme point de vue combiné. Le sélecteur du message n'est alors recherché que dans les morceaux mis en relation par la hiérarchie des morceaux avec *Cinéphile* et *Sportif*, c'est-à-dire que les morceaux *Employé*, *Enseignant* et *Chercheur* ne sont pas examinés lors de cette recherche.

I.3.3.5.3. Discussion

L'approche proposée par Bardou a pour objectif l'intégration de la notion de points de vue dans les langages à prototypes. En effet, dans tout langage à prototypes dans lequel le mécanisme de délégation entraîne le partage de propriétés, il est possible de représenter une entité du domaine selon plusieurs points de vue au sein des représentations éclatées. Sur ce principe, Bardou propose la notion d'objet morcelé dont les propriétés sont définies et détenues par plusieurs morceaux organisés au sein d'une hiérarchie de délégation. Seul l'objet morcelé (et non ses morceaux) a le statut d'objet. Dans cette approche, la notion de point de vue correspond à une combinaison de morceaux spécifiés dans un message. Donc un point de vue est local à un objet morcelé qui utilise un mécanisme de filtrage (tous les points de vue coexistent en même temps) afin de satisfaire les différents messages. Ceci rejoint l'approche VBOOM dans laquelle un point de vue est une combinaison de vues. Cependant, l'approche de Bardou ne propose pas de méthodologie pour élaborer les points de vues pertinents d'un système.

I.3.4. Notion de vue en modélisation

I.3.4.1. Vues en UML

UML (OMG, 2003a) offre des vues de développement pour structurer un système en plusieurs niveaux d'abstraction. Plus précisément, UML supporte le modèle *4+1 vues* (cf. figure 21) proposé par Kruchten (Kruchten, 1995). Ce modèle propose cinq vues : vue des cas d'utilisation, vue logique, vue des processus, vue de réalisation et vue de déploiement. Cependant, les vues d'UML ne sont pas suffisantes pour modéliser finement l'architecture d'un système selon les points de vue des acteurs de ce système. Certes les cas d'utilisation sont utiles dans la phase d'analyse pour modéliser les besoins et les droits d'accès des utilisateurs, mais aucune trace de ces besoins et de ces droits d'accès n'est conservée dans les autres diagrammes et en particulier dans le diagramme de classes.

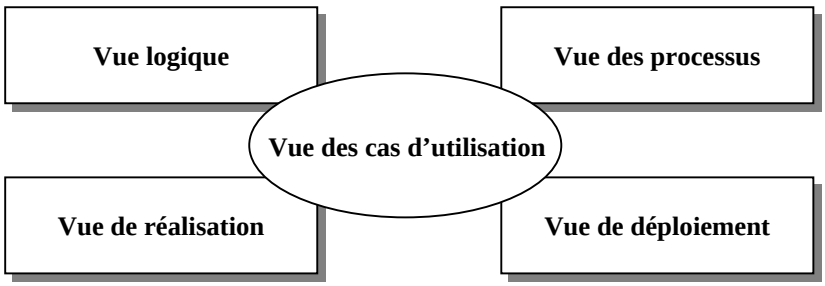

Figure 21 – *Modèle 4+1 vues*

I.3.4.2. Modélisation à base de vues orientées objets

(Motschnig-Pitrik, 2000) propose d'améliorer la notion de vues utilisée dans les bases de données relationnelles en introduisant des vues orientées objets. En effet, les vues dans les bases de données relationnelles ne permettent ni les mises à jour des données ni les insertions de nouvelles informations. Le concept de vue proposé repose sur l'ensemble des opérateurs algébriques ensemblistes de SQL et

de nouveaux opérateurs définis dans (Motschnig-Pitrik, 2000). L'application d'un ensemble de ces opérateurs sur un ensemble de classes dites de base génère des classes dites dérivées. Ces dernières servent pour la personnalisation et l'extension d'un système pour répondre aux besoins et spécificités d'un point de vue donné. Les spécifications d'un *point de vue* peuvent être basées sur différents critères :

- contrôle et droit d'accès ;
- personnalisation du système reflétant les informations pertinentes pour un groupe d'acteurs ou relevant d'une extension particulière du système ;
- une fonction ou une mission particulière.

I.3.4.2.1. Présentation de l'approche

Le but de la solution proposée dans (Motschnig-Pitrik, 2000) est d'aider les développeurs à implémenter des versions personnalisées d'un système complexe en offrant un formalisme de modélisation basée sur les vues. Ce travail est fondé sur une extension d'UML qui introduit un certain nombre de stéréotypes. Cette extension offre les éléments de modélisation suivants :

- *Vue* : terme générique qui désigne un contexte visuel ou une classe *vue*. Elle correspond à une utilisation spécifique du système ;
- *Contexte global* : ensemble des objets de base partageables par toutes les vues ;
- *Classe de base* : classe dont les instances correspondent à des entités stockées. En revanche, une classe dont les informations sont dérivables à partir d'autres via des requêtes OO est dite *classe dérivée* ;
- *Contexte visuel (view context)* : diagramme de classes de base et de classes dérivées.
- *Classe vue* : toute classe composant un contexte visuel est une classe vue.

La dérivation repose sur un ensemble d'opérateurs qui peuvent être synthétisés comme suit :

- les opérateurs ensemblistes usuels (intersection, union, différence) ;
- l'opérateur *select* qui retourne un sous-ensemble selon un prédicat donné ;
- l'opérateur *hide* qui retourne une projection pour extraire des attributs ou des méthodes ;
- l'opérateur *refine* qui permet de dériver de nouvelles informations à partir des informations originelles suite à des calculs ou transformations.

Une vue est donc un treillis de classes à la différence de celle du modèle relationnel qui est une collection plate d'opérateurs algébriques. Par ailleurs, il est possible de mettre à jour les vues orientées objet alors que la mise à jour n'est permise dans une vue relationnelle que si celle-ci contient une clé.

La figure 22 présente un exemple illustratif pris dans (Motschnig-Pitrik, 2000). C'est un diagramme de classes d'un système de gestion administrative d'une université.

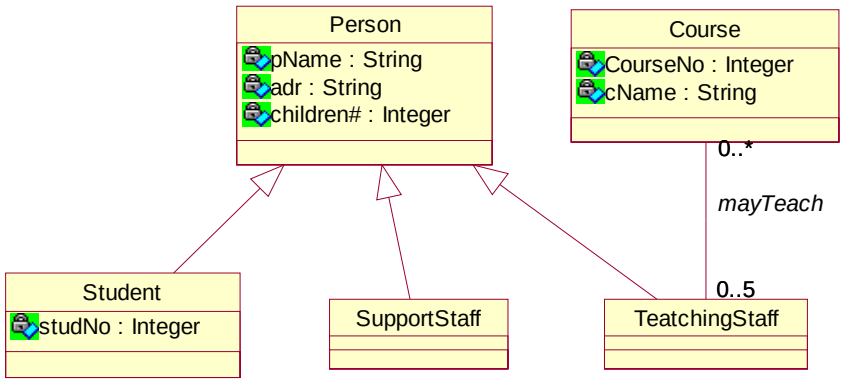

Figure 22 – *Modèle de classes UML représentant un bureau administratif d'une université*

Si l'université décide de mettre en place un bureau d'œuvres sociales visant des activités pour les enfants du personnel, l'administration doit alors détenir un certain nombre d'informations et de statistiques concernant les parents et leurs enfants. Avec la technologie des vues, ce bureau peut avoir un contexte visuel personnalisé assemblant les informations pertinentes. Un ensemble de classes dérivées est défini pour servir cette personnalisation (cf. figure 23). La dérivation est le résultat de l'application d'un ensemble d'opérations algébriques sur les classes de base.

1. Le bureau ne fait pas de différence entre le *supportStaff* et le *TeachingStaff*. Donc une classe dérivée *staff* est définie comme suit :
    a. Staff' := **union**(SupportStaff, TeachingStaff)
    b. Staff := **refine** childStatistics **for** Staff'
2. Une projection est appliquée sur *Staff* pour déduire les informations cruciales concernant le personnel ayant des enfants :
    a. ParentStaff' := **select from** Staff **where** (children #>0)
    b. ParentStaff'' := **refine** parentInfo() **for** parentStaff'
    c. ParentStaff := **refine** kidsBornOn:List **for** ParentStaff

Les classes *Staff* et *ParentStaff* doivent être ajoutées au contexte global. Par ailleurs, Motshnig et al. imposent la classification totale sur n'importe quel contexte à n'importe quel instant. La classification totale induit que le mécanisme d'héritage est exploité au maximum de façon à n'avoir aucune duplication de méthodes ou d'attributs dans un modèle donné.

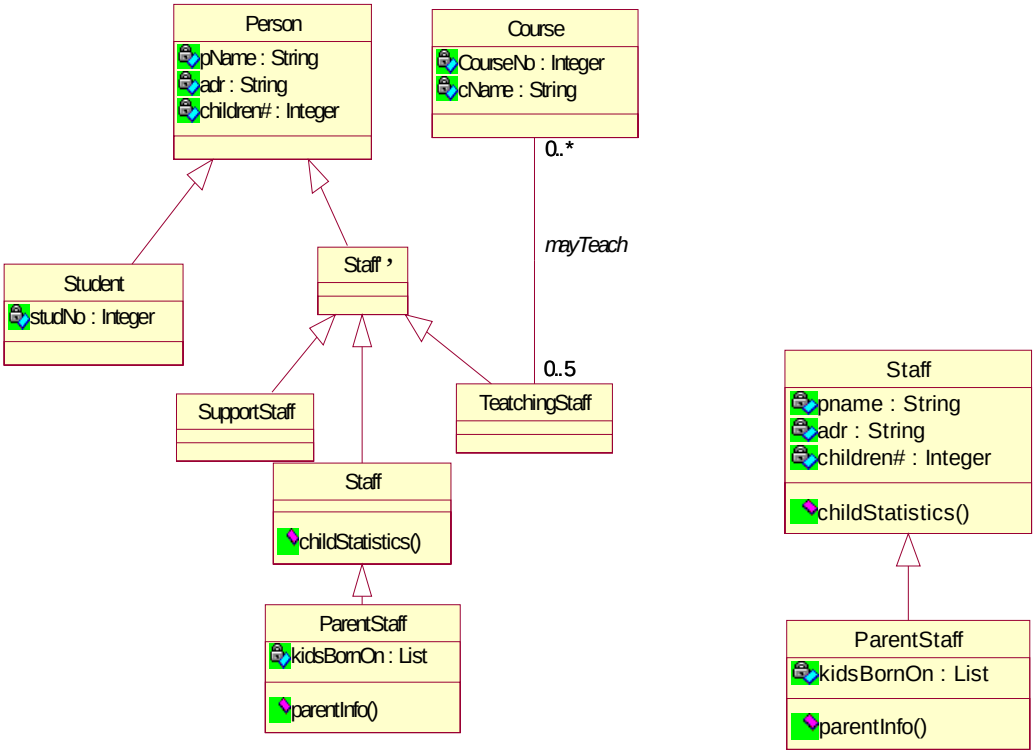

Figure 23 – *Le contexte global (à gauche), le contexte visuel du bureau (à droite)*

Pour pouvoir mener à bien la dérivation de classes et des contextes visuels, Motshnig et al. utilisent un ensemble d'algorithmes de classification automatisée (Rundensteiner, 1992) et d'outils pour aider le développeur à canoniser le système (le rendre totalement classifié) et à extraire des contextes visuels.

I.3.4.2.2. Intégration des vues dans UML

La partie gauche de la figure 24 spécifie la manière dont une classe stéréotypée par « view class » dépend de son élément original de modélisation (class). Chaque classe *vue* est dérivée d'un nombre de classes de base via des opérations algébriques.

Du fait que les packages offrent des espaces de noms séparés pour les éléments qu'ils contiennent, un contexte visuel est modélisé sous forme d'un package stéréotypé. Cette manière de faire permet d'assurer l'indépendance entre les vues. Les packages stéréotypés par « global context » et « view context » (cf. partie droite de la figure 24) sont définis par les contraintes sémantiques suivantes :

- Chaque classe contenue dans le contexte global ou visuel est soit une classe de base soit une classe vue ;
- Aucun changement structurel direct sur les classes du contexte global ou visuel n'est permis.
- Les changements effectués dans le diagramme de classes de base sont répercutés dans le contexte global et les contextes visuels.
- Le contexte global et les contextes visuels sont complètement classifiés.

De plus, les packages stéréotypés par « global context » doivent contenir toutes les classes de bases et les classes vue. D'autre part, chaque package stéréotypé par « view context » doit être dérivé d'un

package stéréotypé par « global context » (un contexte visuel est le résultat de la dérivation d'un contexte global) de sorte que la dérivation produit des sous-graphes du contexte global.

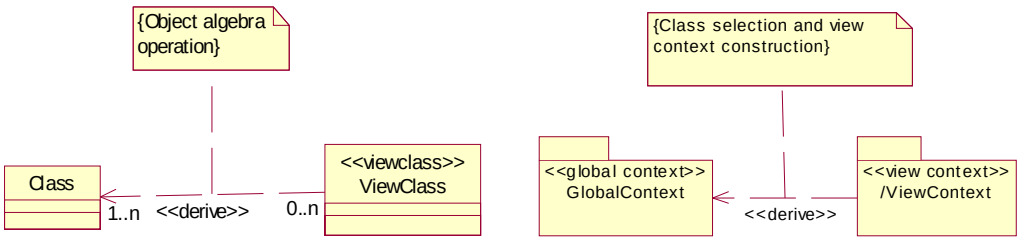

Figure 24 – *Les stéréotypes viewclass, viewContext et globalcontext*

D'autre part, un certain nombre de nouveaux stéréotypes ont été introduits, à savoir : « base » pour désigner les classes de base et « view » pour stéréotyper les classes *vue*. La figure 25 présente le métamodèle proposé par Motchnig et al. qui étend le métamodèle UML afin qu'il supporte leur approche par vues.

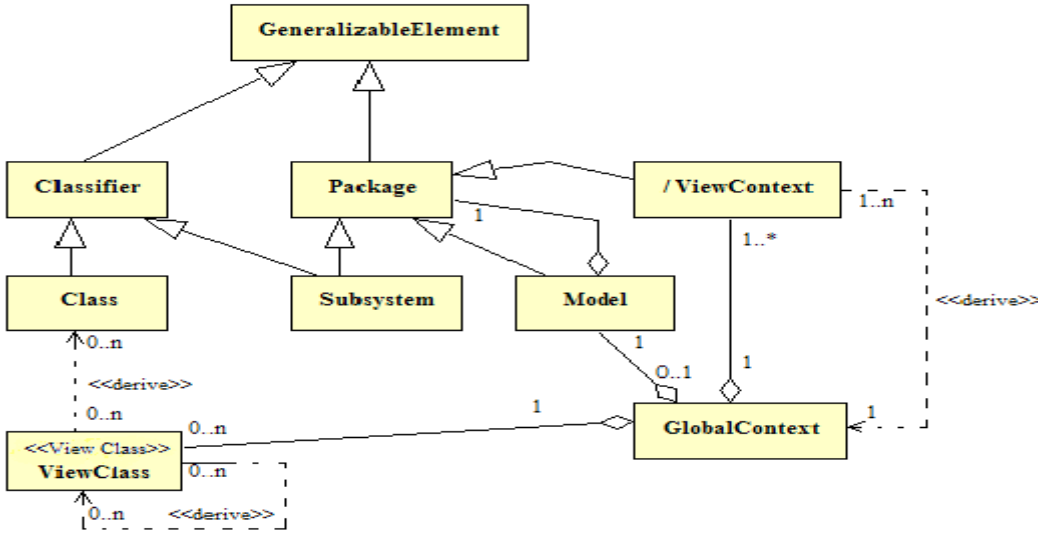

Figure 25 – *Intégration des concepts de modélisation par vues dans le métamodèle UML*

Exemple

Dans cette partie, nous illustrons l'approche de Motchnig et al. par un exemple pris dans (Motschnig-Pitrik et al., 2003). Cet exemple concerne une application serveur qui fait de la création dynamique de pages web et une application cliente qui reçoit et affiche ces pages. Le client n'est pas un navigateur web simple, et la page web n'est pas une page HTML classique, mais elle est enrichie par d'autres données spécifiques. Les pages créées sont stockées dans une base de données du serveur. Un outil de journalisation extrait les pages à partir de la base de données pour des besoins statistiques. Il y a quatre acteurs qui interagissent avec une page web (cf. figure 26). A ces acteurs correspondent quatre vues :

- La vue *serveur* : en plus des données HTML et des données de l'application, le serveur a besoin de connaître l'URL du client.
- La vue *client* : le client a besoin d'afficher la page, donc il doit connaître des informations graphiques.
- La vue *gestionnaire* de la base de données : chaque page est archivée régulièrement.

- La vue *journalisation* : similaire à la vue gestionnaire de la base de données, à part le fait que la journalisation ne doit pas connaître quelles sont les pages qui ont été demandées par un client.

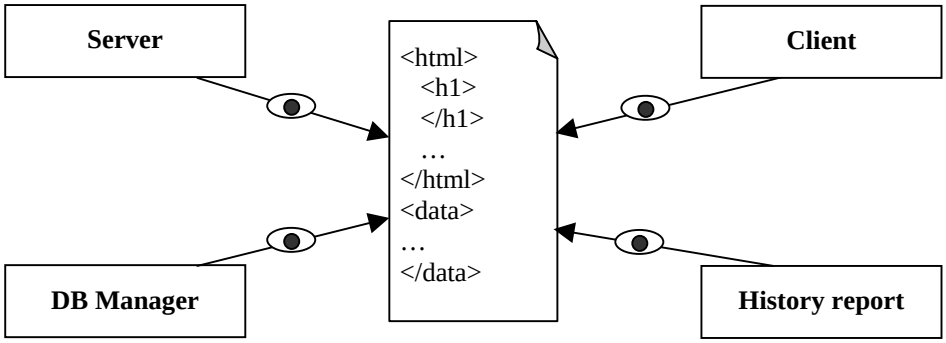

Figure 26 – *Les vues d'une page web*

La figure 27 illustre les classes *ClientPage* et *ServerPage*. A partir de ces classes et en appliquant les opérations de dérivation suivantes on aboutit au contexte global présenté dans la figure 27. La classe stéréotypée par « intermediate » est insérée pour assurer une sémantique correcte et pour maintenir la forme canonique.

PageBase := union(ClientPage, ServerPage)
DBPage := refine [Timestamp] for (ServerPage)
HMPage := hide [ClientInfo] from (DBPage)

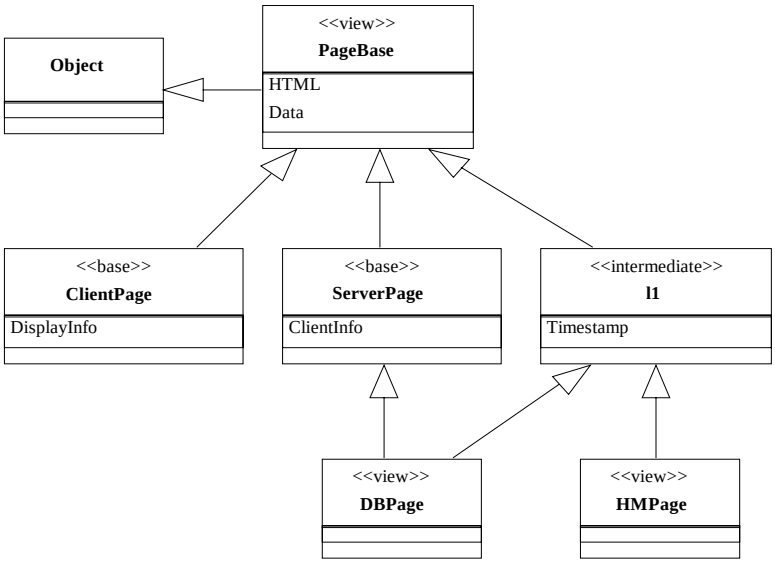

Figure 27 – *Contexte global*

I.3.4.2.3. RoseView, une extension de Rational Rose supportant les vues

Motschnig-Pitrik et al. (Motschnig-Pitrik et al., 2003) ont réalisé l'outil *RoseView* comme enrichissement de *Rational Rose* afin de supporter leur approche. Les algorithmes et les mécanismes utilisés pour implémenter *RoseView* sont basés sur l'approche *MultiView* (Rundensteiner, 1992). Les classes de base sont insérées dans le contexte global en premier lieu puis les classes vue viennent

compléter ce contexte. Le contexte global est la base de l'extraction des vues correspondant à un contexte visuel donné. Cette approche permet de maintenir plusieurs classes vue et plusieurs contextes visuels.

L'interface de *RoseView* offre un *wizard* qui guide l'utilisateur pendant les processus de génération de classes de vue et de dérivation des contextes.
Rational Rose est doté d'un module, appelé *Rose Extensibility Interface (REI)* (Rose-IBM, 2004), qui permet aux applications d'accéder à tous les éléments du modèle *Rose* via des objets *COM*. *RoseView* exploite cette fonctionnalité pour construire et maintenir le contexte global et les contextes visuels dérivés.

I.3.4.2.4. Discussion

Le travail de Motshnig et al. tente d'enrichir la modélisation orientée objet en proposant un mécanisme de point de vue basé sur *MultiView* (Rundensteiner, 1992). L'objectif de cette étude est de fournir des moyens de personnalisation répondant aux besoins particuliers, et des moyens d'extensibilité des systèmes. Elle étend le métamodèle d'UML pour introduire les vues en définissant un certain nombre de stéréotypes sur les classes et les packages. D'autre part, Motchnig et al. ont développé l'outil RoseView en étendant l'atelier *Rational Rose*. Cet outil automatise la construction des vues en guidant le concepteur lors du processus de création des vues. L'approche de Motchnig est plus particulièrement adaptée pour les vues dans les bases de données orientées objet. Par contre, cette approche n'offre aucune démarche à suivre pour élaborer les vues ni aucun moyen pour l'implémentation de ces vues.

I.3.4.3. Modélisation par rôle

Dans un système complexe, un objet interagit avec le monde de manière subjective selon la perspective du contexte de l'utilisateur (Harrison et al., 1993). Par conséquent, les propriétés d'un objet ne sont pas seulement objectives (intrinsèques) mais peuvent être subjectives (extrinsèque). L'origine de l'apparition de la notion de *rôle* est le fait que les propriétés extrinsèques d'un objet peuvent changer dans le temps (Pernici 1990, Kristensen 1995). Un objet peut alors être sujet à des classifications multiples durant son cycle de vie. En effet, "*L'aspect fondamental d'un rôle est qu'il est particulier à une situation. Un rôle est un point de vue temporaire*" (Kristensen et al., 1996a). Pour Riehle et al. (Riehle et al., 1998), un rôle est un type qui décrit la vue qu'a un objet vis à vis d'un autre. Un objet peut jouer différents rôles à un instant donné.

Par ailleurs, cette notion de rôle peut être utilisée pour décrire la participation d'un objet à l'accomplissement d'une activité qui traverse plusieurs objets (Kristensen et al. 1996b, Andersen 1997).

Kristensen et al. (Kristensen et al., 1996a) considèrent un rôle comme un ensemble de propriétés extrinsèques qui sont attribuées à un objet pour jouer un rôle selon une perspective déterminée. Riehle et al. (Riehle et al., 1998), quant à eux, distinguent les propriétés intrinsèques de celles qui sont comportementales. Les propriétés intrinsèques sont modélisées dans un espace d'états abstrait alors que les propriétés comportementales sont modélisées par les différents types de rôle.

Dans cette section nous présentons les différents concepts de l'approche par rôle, puis la méthodologie par rôle OOram.

I.3.4.3.1. Concepts principaux de l'approche par rôle

La notion de rôle proposée par (Kristensen et al., 1996a) introduit un nouveau type d'abstraction appelé "*Roleification*" qui diffère des concepts traditionnels de classification et généralisation. La *roleification* est en fait caractérisée comme suit :
Classification dynamique : l'attribution de différents rôles durant le cycle de vie d'un objet lui permet de changer sa classification dans le temps.
Identité unique : l'identité d'un sujet (objet avec son rôle effectif) est une identité unique attribuée à l'objet durant tout son cycle de vie.
Extension : un rôle ne modifie pas les propriétés intrinsèques d'un objet, mais, il en ajoute de nouvelles.
Une dépendance : un rôle n'a d'existence que pour être "joué" par un objet.
La multiplicité : un objet peut jouer plusieurs rôles en même temps.
L'abstraction : les rôles peuvent être classifiés et organisés sous forme de hiérarchie.
L'agrégation : un rôle peut être composé par plusieurs rôles.

Rôle et activité : Les travaux de Kristensen et al. (Kristensen et al., 1996b) et ceux d'Andersen et al. (Andersen et al. 1992, Andersen 1997) considèrent à la fois le rôle et l'activité. Pour Kristensen, une activité transversale est une activité dont l'exécution fait participer plusieurs objets. Kristensen propose l'activité comme un mécanisme d'abstraction qui permet de relever les interactions inter-objets durant l'accomplissement d'une tâche donnée. Une activité est donc considérée comme une entité définie par un ensemble de participants et une directive qui détermine le comportement des différents participants (collective behavior). Une activité peut être spécialisée ou composée. Un objet qui participe à l'accomplissement de plusieurs activités instancie différents rôles.
Rôle et classe : Pour un objet appartenant à une classe, un rôle décrit les propriétés pertinentes pour la réalisation de l'activité concernée (Kristensen et al., 1996b) alors que la classe décrit les objets indépendamment d'un environnement d'interaction (Andersen 1997). Une classe décrit donc un objet pour la totalité de son cycle de vie (verticalement à toutes les activités dans lesquelles l'objet est candidat à participer).

La même définition de classe est reprise par (Riehle, 2000). En effet, Riehle et al. définissent une classe comme un ensemble de rôles que ses différentes instances peuvent jouer. L'union de toutes les opérations et les propriétés définies par les différents rôles de ces instances constituent l'interface d'une classe. Une classe combine alors les différents rôles d'un objet et le modèle de classe combine les différents modèles de rôle.

I.3.4.3.2. OOram

(Andersen 1997) décrit les objets participant à la réalisation d'une activité et les interactions inter-objets pour son accomplissement par un modèle de rôle.

Riehle et al. (Riehle et al., 1998) définissent aussi le modèle de rôle comme un espace de noms dans lequel est regroupé un ensemble de rôles d'objet correspondant à une perspective déterminée. Dans OOram (Reenskaug, 1995), un modèle de rôle contient les entités et les comportements qui relèvent d'une collaboration particulière. L'architecture en 3-modèles de (Reenskaug, 1997) supporte la définition des systèmes couvrant le cycle de développement de logiciels. Cette architecture est basée sur la définition de trois types de modèles : modèle d'entreprise, modèle d'information et le "model services". Le premier est utilisé pour identifier et définir les rôles joués dans une organisation, le

deuxième décrit les informations gérées par le système tandis que le troisième montre les interfaces homme/machine.

I.3.4.3.3. Discussion

La notion de rôle permet une flexibilité d'adaptation d'objets pour répondre à des attentes d'autres objets ou pour participer à l'accomplissement d'une activité donnée. L'attribution de rôle à un objet lui permet une évolution dynamique dans le temps tout en gardant une identité unique. Un rôle est une extension d'une classe d'objet. Les propriétés d'un rôle sont des propriétés extrinsèques requises pour jouer un rôle dans une situation déterminée. Cependant, cette notion de rôle ne permet ni le filtrage d'informations pertinentes pour un acteur ni le contrôle des droits d'accès.

I.4. Etude comparative des différentes approches par point de vue le long du cycle de vie d'un logiciel

Comme nous l'avons vu dans les sections précédentes, plusieurs approches ont abordé la notion de point de vue sous différents termes : sujet, rôle, aspect, vue, etc. Les avantages principaux des points de vue sont : (i) la réduction de la complexité du développement ; (ii) l'amélioration de la pertinence des informations et la gestion des droits d'accès selon les profils des utilisateurs.

Notre étude comparative vient compléter un ensemble d'études similaires déjà effectuées par différents auteurs : (Bardou et al., 1998), (Bendelloul et al., 2000) et (Hanenberg et al., 2002). En effet, l'étude comparative proposée ici et détaillée dans (El Asri et al., 2004) traite plus particulièrement de la manière dont les approches orientées point de vue influence le cycle de vie d'un logiciel. Ainsi, la comparaison concerne les deux phases principales du cycle de vie de la production des logiciels que sont **le développement** et **l'exploitation**. Cette comparaison porte sur un ensemble de critères que nous avons jugés parmi les plus pertinents pour ces deux phases.

I.4.1. Critères de développement

Pour la phase de développement, nous avons retenu les critères suivants : l'utilisation des points de vues au long du processus du développement, la réutilisation, la facilité de compréhension du code, et la testabilité.

***L'utilisation des points de vues le long du processus du développement* :** L'utilisation des points de vues le long d'un processus de développement est un facteur essentiel. En effet, il permet une focalisation profonde sur les différents domaines et métiers liés au système. Par ailleurs un développement par point de vue améliore la traçabilité et la compréhension des livrables (documents, codes, etc.). Nous avons donc jugé utile d'étudier l'intégration du concept de point de vue par les différentes approches dans les trois sous-phases d'analyse, de conception et d'implémentation.

Sous-phase d'analyse : L'intégration de la notion de point de vue est explicitement prise en charge par la méthode par rôle OOram. La méthodologie OOram consiste à identifier en premier lieu pendant la phase d'analyse l'ensemble des champs de préoccupation. Cette identification est une description précise des différents phénomènes fonctionnels, technologiques et métiers d'un système.

Sous-phase de conception : l'utilisation des points de vue lors de la sous-phase de conception est supportée - de différentes manières - par les différentes approches.

La méthodologie par rôle OOram consiste à élaborer un ensemble de modèles de rôle. Ces modèles sont présentés sous forme de modèles de collaboration, d'interfaces ou de sémantique. Ces modèles décrivent les différents rôles joués par les acteurs et les données d'un système d'une part et les interactions entre ces rôles d'autre part. La synthèse et la composition d'un ensemble de modèles édifient un système du monde réel (Reenskaug, 1999).

Pour l'approche par sujet, Clark et al. (Clark et al., 1999) proposent de décrire un ensemble de propriétés concernant une perspective dans un modèle de sujet. L'intégration d'un ensemble de modèles de sujet compose un système réel. La conception MDSOC (Ossher et al., 2001) consiste à décrire les différentes préoccupations d'un système dans des hyperslices dans un espace multidimensionnel. La composition d'un ensemble d'hyperslices dans un hypermodule bâtit un système personnalisé selon les préoccupations requises.

La conception par vues proposée par Mili et al. (Mili et al., 2000) utilise les concepts classiques d'agrégation et d'association pour modéliser un objet d'application. Un objet d'application se compose d'un objet de base et d'un ensemble de vues qu'on peut activer ou désactiver de manière dynamique. A la conception, les vues sont des objets liés avec l'objet de base par la relation d'agrégation.

Enfin, pour l'approche par aspect, Suzuki et al. (Suzuki et al., 1999) proposent un ensemble de stéréotypes pour exprimer les classes d'aspects et les opérations de tissage.

Sous-phase d'implémentation : les différentes approches par points de vue proposent des supports pour le codage. Nous distinguons d'abord celles qui proposent de nouveaux paradigmes, outils ou langages pour le support des concepts de vue comme : SOP et Hyper/J pour l'approche par sujet ; AspectJ pour l'approche par aspect et le préprocesseur défini par Mili et al. En deuxième lieu, nous distinguons celles qui proposent des *mappages* vers des langages de programmation existant comme l'approche par rôle.

La réutilisabilité **:** c'est un critère de qualité clé qui concerne toutes les sous-phases du développement. Ce critère est assuré par les approches de différentes manières.

La synthèse et la dérivation des modèles de rôle par OOram permettent de réutiliser la modélisation d'un framework pour un métier déterminé. Le mécanisme de composition proposé par l'approche par sujet permet la réutilisation des hyperslices. L'approche par aspect permet d'encapsuler les exigences non fonctionnelles d'un système sous forme d'aspects ce qui permet leur réutilisation par un ensemble de composants.

Intelligibilité du code **:** Le processus d'évolution et de mise à jour des systèmes repose en grande partie sur la capacité de compréhension de codes existants. Cette tâche prend un temps considérable dans les actions de réparation de bug et d'intégration de nouveaux besoins. Un code intelligible et propre est un élément déterminant pour l'évolution de tout système. Les modélisations par rôle, sujet, vue ou aspect - de part la séparation des préoccupations qu'elles induisent - permettent l'écriture de code clair et compréhensible.

Testabilité **:** La séparation des préoccupations par les différentes approches étudiées permet aux différents clients avec leurs savoir-faire de se focaliser sur leurs domaines d'intérêts. Cette focalisation facilite les tests unitaires aussi bien fonctionnels (en boîte noire) que structurels (en boîte blanche). En effet, le programmeur qui effectue des tests unitaires est limité à son domaine, et donc une bonne compréhension du code lui facilitera notamment les tests en boîte blanche.

I.4.2. Critères d'utilisation

Pour les critères d'utilisation nous avons identifié les critères suivants : profilage et gestion des droits d'accès ; dynamisme ; multiplicité ; intégrité et maintenabilité.

***Profilage et gestion des droits d'accès* :** Nous pouvons définir le profilage comme une manière d'assurer la pertinence d'information et la gestion des droits d'accès des utilisateurs d'un système. Chaque acteur possède un profil qui détermine les données valides pour lui, ses spécifications de visualisation, etc. Le concept de point de vue, défini par les différentes approches, est un moyen trivial de profilage. Les systèmes sont analysés, conçus et codés selon les différentes perspectives. L'approche par aspect est une exception. En effet, cette approche est un moyen de séparation de préoccupations non fonctionnelles. Par conséquent, elle ne permet ni le profilage ni la gestion des droits d'accès.

***Dynamisme* :** Les mécanismes d'activation/désactivation proposés par Mili et al. (Mili et al., 2000) permettent l'évolution dynamique des profils. Reeskaug et al. (Reensckaug, 2001) proposent le schéma OOCS et un configurateur temps-réel pour changer dynamiquement le rôle d'un objet. L'approche par sujet quant à elle ne permet pas l'évolution dynamique de l'univers d'un système. En effet, la composition d'un ensemble de profils est faite au moment de la compilation.

***Multiplicité* :** Dans un système distribué, plusieurs acteurs avec des profils différents peuvent accéder simultanément aux même objets. Mili et al. (Mili et al., 2002) proposent un outil DOC (Distributed Object Configurator) pour choisir l'ensemble des vues actives d'un objet d'application et gérer l'aiguillage des méthodes vers les vues actives à la manière de l'approche par sujet. L'approche par rôle permet à un objet de jouer plusieurs rôles simultanément. Cette multiplicité est gérée selon le contexte de l'objet.

***Identité et Intégrité* :** Profils, contextes et perspectives partagent le même ensemble d'objets (différemment classifiés selon les différents types de points de vue). L'approche par rôle et l'approche par sujet considèrent les vues comme des extensions de l'objet de base qui ne détiennent aucune identité. Par contre l'approche proposée par Mili et al. considère les vues comme des composants de l'objet de base. Les aspects, quant à eux, ne sont pas des composants, mais des modules qui traversent un ensemble d'objets.

***Maintenabilité* :** L'ingénierie des systèmes consiste en grande partie à traiter l'évolution et la maintenance des systèmes. L'évolution des systèmes est favorisée par l'application du concept de point de vue. En effet, l'addition de nouveaux besoins d'un acteur ou de nouvelles préoccupations se fait de manière non-invasive du moment où les préoccupations sont séparées.

I.4.3. Tableaux récapitulatifs

Nous résumons notre étude comparative dans les deux tableaux présentés dans les figures suivantes. Le tableau de la figure 28 propose une évaluation des critères de développement pour les principales approches étudiées dans ce chapitre. Le tableau de la figure 29 se focalise sur les critères d'exploitation. Pour les deux tableaux, nous avons utilisé des étoiles '*' qui représentent le degré de qualité de chaque critère.

	Utilisation du concept de point de vue le long du processus de développement			Réutilisabilité			Intelligibilité de code	Testabilité
	Analyse	*Conception*	*Implémentation*	*Analyse*	*Conception*	*Implémentation*		
SOP	-	***	***	-	***	**	**	**
Rôles	**	**	*	-	**	*	**	**
VP	-	***	**	-	***	**	**	**
AOP	-	*	***	-	**	***	***	**

Figure 28 – *Tableau récapitulatif synthétisant les critères de développement*

	Profilage et gestion des droits d'accès	Dynamisme	Multiplicité	Identité et Intégrité	Maintenabilité
SOP	**	-	***	***	**
Rôles	***	*	***	***	*
VP	***	***	***	**	**
AOP	-	*	-	-	**

Figure 29 – *Tableau récapitulatif synthétisant les critères d'utilisation*

Légende : *** : fortement ; ** : moyennement ; * : faiblement ; - : Non supporté ou n'est pas décrit dans la littérature

Abréviation : SOP : Subject-Oriented Programming ; VP : View Programming ; AOP : Aspect-Oriented Programming

I.5. Conclusion

Dans ce chapitre, nous avons présenté en premier lieu la problématique posée par la modélisation des systèmes complexes. Nous avons examiné ensuite l'existant en terme d'approches prenant en compte le concept de point de vue, d'une manière directe ou indirecte.

Cette étude nous a convaincu de la nécessité d'élaborer une méthodologie intégrant pleinement l'approche multivues dans la modélisation par objet. Cette approche doit principalement : (i) être facile à mettre en oeuvre en limitant le non déterminisme inhérent à plusieurs approches mentionnées, (ii) réutiliser les standards actuels issus de l'OMG et (iii) cibler les langages à objet du marché.

Notre objectif est ainsi de réaliser un environnement de définition, de construction et d'utilisation des classes, vues et points de vue. Nous avons décidé pour cela d'étendre le standard UML sous forme d'un profil, de proposer une démarche associée dirigée par les modèles qui s'intègre dans le cadre MDA (Model Driven Architecture), et de fournir des moyens de génération de code multi-cibles via des patrons d'implémentation.

Chapitre II

Approche VUML

II.1. Introduction

De nos jours, chacun s'accorde à reconnaître l'intérêt de prendre en compte l'utilisateur - au sens large du terme - le plus tôt possible dans le développement des systèmes complexes. Le concept de point de vue est un moyen pertinent pour mettre en oeuvre cette approche. Parmi les bénéfices de l'approche, on peut citer l'utilisation d'un modèle unifié avec points de vue multiples au lieu de plusieurs sous-modèles interdépendants et souvent incohérents, l'instanciation par point de vue, le changement dynamique de point de vue, la gestion de la cohérence entre sous-modèles, etc. (Coulette et al., 1996).

Pour ces raisons, nous pensons que le développement, la maintenance et la réutilisation de systèmes d'information basés sur les vues des utilisateurs (observatoires multivues numériques) vont jouer un rôle stratégique dans le futur. Pour développer de tels systèmes, nous avons besoin d'une méthodologie qui supporte explicitement – de l'analyse à l'implémentation – les concepts d'acteur, de vue et de point de vue.

Comme nous l'avons vu dans l'état de l'art (cf. chapitre I), plusieurs approches par point de vue ont été proposées. Dans ce contexte, un certain nombre de travaux de recherche ont été menés par notre équipe au sein du projet VBOOM (cf. section I.2 du chapitre I). Ces travaux ont visé l'introduction du concept de vue et point de vue dans la modélisation par objets. Le modélisateur d'un système complexe (par exemple un satellite) doit pouvoir définir des vues multiples sur ce système, puis accéder à l'une ou à plusieurs de ces vues selon ses besoins particuliers. Pour cela, un nouveau concept « classe flexible », et une nouvelle relation appelée « visibilité », intégrés au sein d'une extension d'Eiffel appelée VBOOL (View Based Object-Oriented Language) (Marcaillou, 1995), ont été définis. Dans le même contexte, une méthode d'analyse et de conception par objet VBOOM (View Based Object Oriented Method) (Kriouile, 1995) supportant les concepts de vue et point de vue a été élaborée. Cette méthode permet de modéliser un système selon les points de vue de ses différents utilisateurs. Cependant, comme nous l'avons déjà mentionné dans le chapitre I, le langage cible de la méthode VBOOM, appelé VBOOL, reste un langage orienté objet complexe. De plus, l'élaboration et la gestion des vues dans VBOOM posent plusieurs difficultés d'ordre méthodologique (cf. section I.2.3. du chapitre I).

Le langage UML (Unified Modelling Language) (OMG, 2001) marque une évolution certaine des approches objets utilisables à différents niveaux d'abstraction et phases du développement de logiciels. UML et les AGLs associés offrent la notion de vues de développement (cas d'utilisation, logique, déploiement…) qui sont utiles pour structurer et décomposer un système en plusieurs niveaux d'abstraction, mais les vues dans UML sont insuffisantes pour modéliser l'architecture d'un système selon les points de vue des utilisateurs. Les cas d'utilisation sont très utiles durant la phase de l'analyse

mais aucune trace explicite des points de vue des utilisateurs ne sera gardée dans le diagramme de classes.

Forts de l'expérience du projet VBOOM, nous avons décidé de situer notre approche dans le contexte d'UML et des outils supports du marché, en ciblant les principaux langages à objet.

Ce chapitre a pour but de présenter globalement notre approche de développement multivues, appelée VUML (View based Unified Modeling Language), dont le noyau est un profil UML (Nassar et al. 2003, Nassar 2003, Nassar et al. 2005, Nassar 2004). L'élaboration d'une telle approche se justifie par l'absence d'une approche qui intègre les notions de vue et de point de vue de l'analyse jusqu'à l'implémentation en passant par la conception (El Asri et al., 2004).

VUML est un profil UML basé sur les vues. Ce profil offre un formalisme étendant celui d'UML et une démarche inspirée de celle de la méthode VBOOM (Nassar, 2004). L'ajout principal à UML est le concept de *classe multivues* qui est composée d'une base (partie partagée par tous les acteurs) et d'un ensemble de vues (extensions de la base). A la différence de VBOOM, chaque vue correspond aux besoins spécifiques d'un acteur (utilisateur final, développeur, ...) c'est-à-dire au point de vue de cet acteur sur l'entité considérée. VUML offre des mécanismes pour gérer les droits d'accès aux classes multivues, spécialiser une classe multivues, spécifier les dépendances entre les vues, assurer la cohérence du modèle en cas de mises à jour, administrer les vues à l'exécution, etc. En plus du concept de classe multivues, VUML introduit aussi le concept de *composant multivues* permettant de représenter une classe multivues au niveau du diagramme des composants. La spécificité d'un composant multivues est la possibilité d'avoir des interfaces dont l'accessibilité et le comportement changent dynamiquement selon le point de vue de l'utilisateur courant du système. La sémantique de VUML est partiellement décrite moyennant une extension du métamodèle UML ; elle sera détaillée et formalisée en OCL dans le chapitre suivant.

En plus des mécanismes et concepts VUML, nous proposons un patron d'implémentation permettant de générer du code objet multi-cibles (Java, C++, Eiffel ,…) à partir d'une modélisation VUML.

Le reste de ce chapitre est structuré de la manière suivante. La section II.2 décrit les limites d'UML en ce qui concerne la gestion des droits d'accès aux informations. Dans la section II.3, nous abordons le concept de classe multivues et les mécanismes qui lui sont liés. Puis nous présentons dans la section II.4 notre approche pour intégrer la notion de point de vue dans les composants logiciels. La section II.5 consiste en une présentation de l'aspect dynamique de notre approche et en particulier la description d'un patron de génération de code multi-cibles à partir d'une modélisation VUML. La dernière section décrit le profil proposé pour spécialiser UML afin qu'il supporte l'approche VUML.

II.2. Limites d'UML pour la gestion des droits d'accès

Dans cette section, nous montrons les limites d'UML en ce qui concerne la prise en compte des droits d'accès aux informations. Nous illustrons cette partie et notre approche par la suite à travers un système de gestion d'un établissement concessionnaire de voitures (neuves et d'occasion). La figure 30 montre un extrait du diagramme de cas d'utilisation de ce système. Les acteurs du système sont : Directeur de l'établissement, Responsable publicité, Commercial, Maintenicien (Mécanicien, Electricien, Carrossier-tôlier), Client, Agent de police.

Activités des acteurs

- Le directeur supervise toutes les activités de l'établissement afin de produire des bilans et prendre des décisions pour améliorer le rendement de cet établissement.
- Le responsable publicité traite tout ce qui concerne la publicité.
- Le commercial s'occupe de l'approvisionnement en voitures, de la négociation des prix avec les clients et la vente des voitures.
- Le maintenicien entretient les voitures.
- Le client interagit avec le système afin de consulter les offres disponibles, voir les promotions, négocier les prix et commander des voitures.
- L'agent de police enregistre les accidents concernant les voitures.

Seul un sous-ensemble des cas d'utilisation identifiés est considéré dans la figure 30 : supervision, gestion des voitures, gestion des ventes, approvisionnement, publicité, finance.

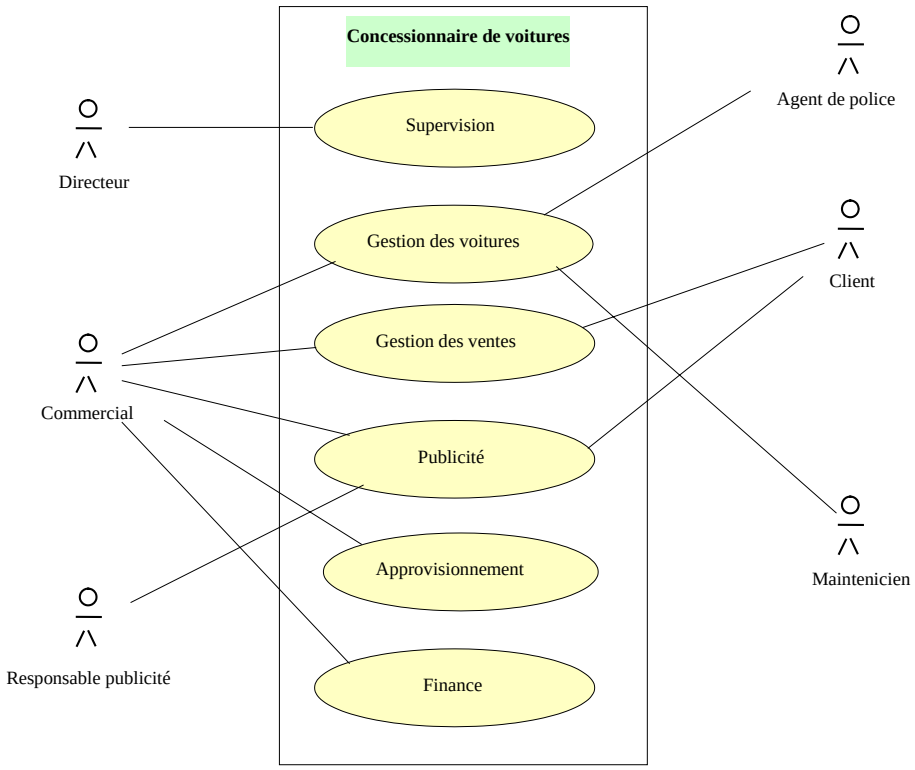

Figure 30 – *Extrait du diagramme des cas d'utilisation du système "concessionnaire de voitures"*

Diagramme de classe UML

Les entités principales du système « Concessionnaire de voitures » sont : *Voiture*, *Moteur*, *BoiteAVitesses*, *Batterie*, *Carrosserie*, *Constructeur*, *Accident*, *Panne*, *Maintenicien*, *Mécanicien*, *Electricien*, *Carrossier-tôlier*, *Catalogue*, *Commande*, *Facture*, etc. Pour simplifier l'illustration des différents concepts introduits par VUML, nous allons nous focaliser sur l'entité « Voiture ». Suite à une analyse/conception réalisée en UML on considère qu'une voiture est caractérisée par une référence, une marque, une couleur, un carburant, une consommation, un historique – liste des pannes et liste des accidents (pour les voitures d'occasion) –, un prix de vente, un prix conseillé, une remise, et un constructeur. Elle est composée, pour simplifier, d'un moteur, d'une boîte à vitesses, d'une

batterie, d'une signalisation, et d'une carrosserie. Le système permet à un client de consulter les informations concernant les voitures et d'acheter des voitures. Il permet aussi au commercial de mettre à jour les prix et les remises. En ce qui concerne la maintenance des voitures, le maintenicien a la possibilité de consulter et de mettre à jour l'historique des pannes ; de plus il a accès à un certain nombre d'informations techniques. La figure 31 ci-dessous présente un extrait du diagramme de classes UML du système « Concessionnaire de voitures ».

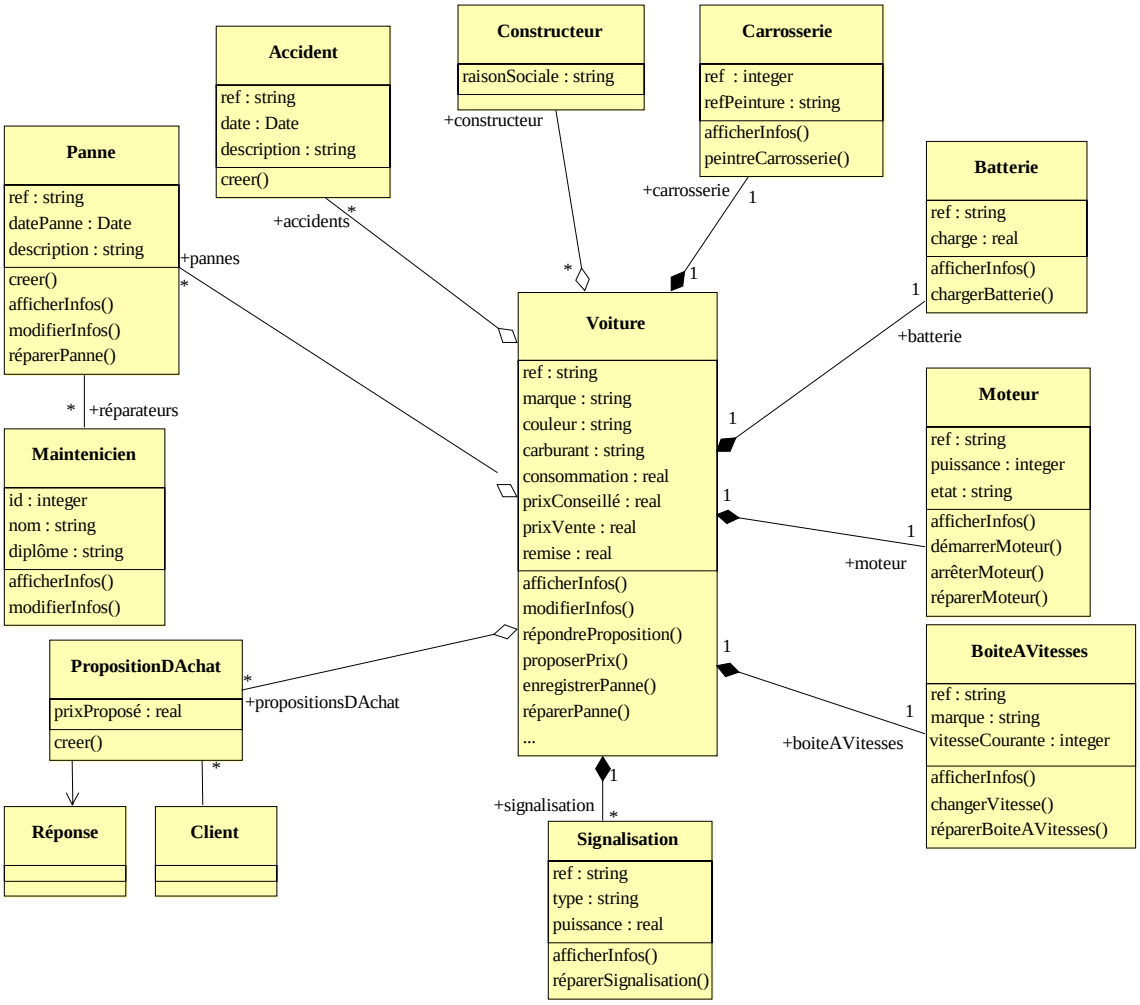

Figure 31 – *Diagramme de classes UML du système « Concessionnaire de voitures » (extrait)*

Discussion

Dans un diagramme de classes UML, chaque acteur a potentiellement les mêmes droits d'accès aux informations et aux services encapsulés dans les classes. Cependant, les acteurs n'ont pas les mêmes besoins et les mêmes responsabilités. A titre d'exemple, dans le diagramme de classes de la figure 31, le client peut accéder à toutes les informations concernant la vente des voitures (prix conseillé, prix de vente, remise) et à tout l'historique des pannes. Ceci n'est pas acceptable car une partie des informations de vente des voitures doit être cachée au client, et certaines informations de l'historique ne doivent être accédées que par le maintenicien.

Dans UML, le contrôle des droits d'accès ne peut pas être réalisé dans le diagramme de classes mais seulement dans les diagrammes dynamiques (diagramme de séquence, diagramme de collaboration,…), et dans ce cas le contrôle doit être programmé dans les méthodes des classes. Notre objectif est de décrire ces droits d'accès aux informations à un niveau d'abstraction élevé, c'est-à-dire lors de la phase d'analyse.

Afin d'atteindre ce but, nous avons décidé d'introduire un nouveau type de classe, la *classe multivues*, qui permet de définir les vues associées aux profils des acteurs. Le défi est de placer judicieusement les informations (attributs, méthodes, contraintes) dans les vues d'une classe donnée, ce qui permet de définir des droits d'accès. Le principe consiste donc à associer un point de vue à chaque acteur et à lui affecter des droits d'accès aux informations offertes par la classe tout en ayant la possibilité de passer d'un point de vue à un autre d'une manière dynamique. Comme nous allons le voir dans la section suivante, les vues ne peuvent pas être modélisées comme des classes descendantes. Nous avons besoin d'une nouvelle relation de dépendance que nous appelons *viewExtension*. Le concept de classe multivues et les mécanismes associés constituent le noyau de notre profil UML présenté plus loin dans ce chapitre.

II.3. Concept de classe multivues

Le concept de classe multivues est l'élément clé de l'approche VUML. Il constitue le principal ajout à UML. Par rapport à une classe "habituelle", une classe multivues est dotée d'une ou plusieurs vues. Dans cette section, nous présentons tout d'abord les principes de base de VUML, puis nous donnons quelques définitions relatives aux concepts introduits dans VUML. Ensuite nous décrivons la structure des classes multivues et nous détaillons les mécanismes associés.

II.3.1. Principes et définitions

Comme nous l'avons mentionné dans le chapitre I, la notion de vue proposée dans UML ne correspond pas à nos besoins. UML n'offre pas de mécanisme de granularité fine pour modéliser les points de vue des acteurs tout au long du développement. En effet, si les cas d'utilisation et les diagrammes de séquence UML sont des moyens efficaces pour modéliser les besoins et les droits d'accès des acteurs, le diagramme de classes — qui joue un rôle central dans la modélisation — n'en garde pas de trace. Au contraire VUML est fondée sur une approche centrée utilisateur, dont l'objectif est de représenter les besoins et les droits d'accès des utilisateurs de l'analyse jusqu'à l'implémentation.

VUML conserve le même objectif d'une modélisation multivues à granularité fine que celui de VBOOM, mais en adoptant les principes suivants :
- Association déterministe d'un point de vue à chaque type d'acteur (utilisateur final ou non) ;
- Association d'une vue unique à chaque point de vue sur un objet donné ;
- Implantation et gestion de l'évolution des vues par l'intermédiaire de la délégation (au lieu de l'héritage multiple).

Nous définissons informellement les concepts clés de VUML comme suit :
Acteur : entité logique ou physique qui interagit avec le système.
Point de vue : vision d'un acteur sur le système (ou sur une partie de ce système).

Vue : entité de modélisation (statique). Elle correspond à l'application d'un point de vue à une entité donnée.

Par simplification de langage, nous dirons dans la suite qu'une vue est associée à un acteur en considérant comme implicite l'entité sur laquelle le point de vue de l'acteur s'applique.

Une **classe multivues** est une unité d'abstraction et d'encapsulation composée d'une **base** (partie commune accessible par tous les acteurs de la classe multivues) et d'un ensemble de **vues** (étendant cette partie commune) représentant les besoins et les droits d'accès des acteurs. Chaque vue correspond à un seul acteur. Au moment de l'exécution, un **objet multivues** est une instance d'une classe multivues concrète. Une seule de ses vues est active, et correspond à un **point de vue**, qui est le point de vue de l'utilisateur courant du système (Nassar et al., 2003a).

Une **classe publique** est une classe accessible par tous les acteurs du système.

II.3.2. Structure statique d'une classe multivues

La figure 32 ci-dessous illustre la structure statique d'une classe multivues. Une telle classe est composée d'une base (stéréotype « *base* ») qui a le même nom que la classe UML correspondante, et des vues (stéréotype « *view* ») qui sont reliées à la base via une relation *viewExtension*. L'activation d'une vue (liaison avec le point de vue de l'utilisateur courant) est faite lors de l'exécution. La relation de dépendance *viewExtension* n'est pas une relation d'héritage : les vues dépendent de la base au sens où les attributs et les méthodes de la base sont implicitement partagés par les vues de la classe multivues ; en outre, une caractéristique de vue peut redéfinir une caractéristique de la base.

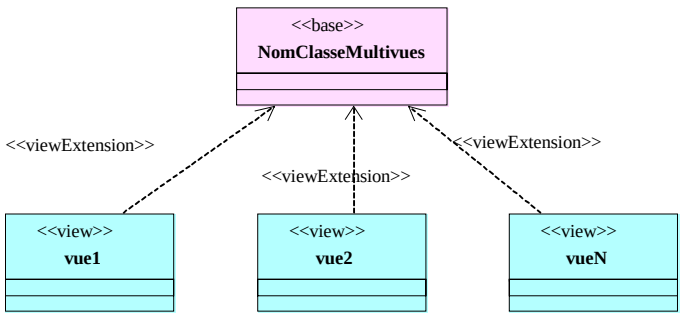

Figure 32 – *Structure statique d'une classe multivues*

La figure 33 ci-dessous montre le diagramme de classes VUML correspondant au diagramme de classes UML de la figure 31.

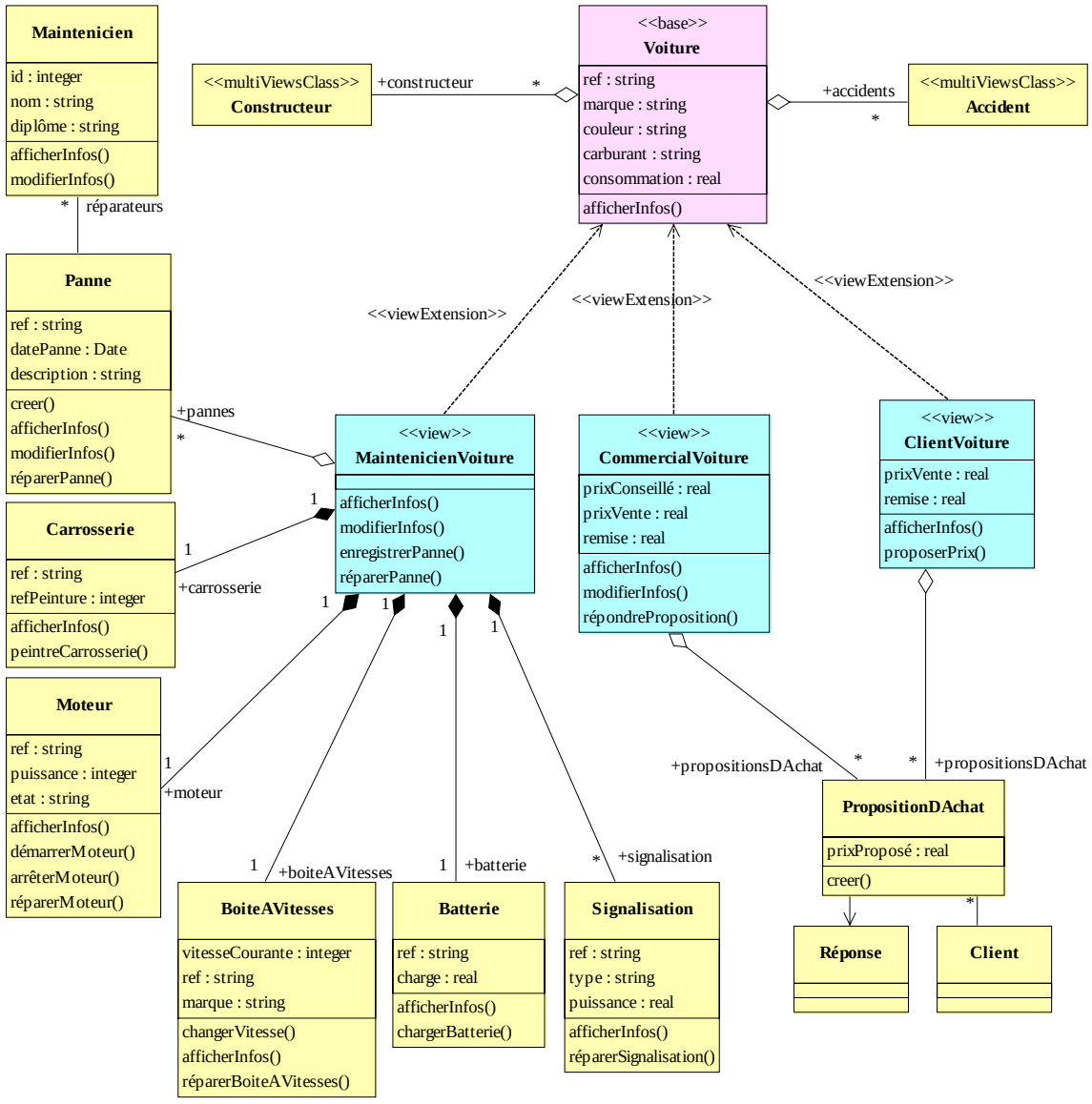

Figure 33 – *Modèle VUML du système "Concessionnaire de voitures" (simplifié)*

Nous pouvons remarquer que chaque classe du modèle UML peut devenir une classe multivues, selon les droits d'accès aux informations des acteurs sur cette classe. Dans ce modèle VUML simplifié, nous spécifions seulement 3 vues associées aux acteurs client, commercial et maintenicien. La classe multivues *Voiture* est constituée d'une base et de trois vues correspondant aux points de vue de ces trois acteurs. La base contient les informations (attributs et méthodes) partagées par ces acteurs. La classe *CommercialVoiture* décrit la spécificité d'une voiture selon le point de vue d'un commercial, tandis que la classe *MaintenicienVoiture* décrit les spécificités d'une voiture selon le point de vue d'un

maintenicien, alors que la classe *ClientVoiture* décrit les spécificités d'une voiture selon le point de vue d'un client. On peut remarquer que la méthode *afficherInfos()* de la base est redéfinie dans les trois vues. Toutes les classes d'un système sont potentiellement multivues car on peut toujours ajouter un point de vue au modèle.

Par rapport au diagramme de classes UML, la distribution des attributs et des méthodes sur les classes est changée. Nous avons défini une seule classe multivues dans ce modèle simplifié – Voiture – mais d'autres classes peuvent aussi être des classes multivues : *Moteur, BoiteAVitesses, Carrosserie, Signalisation, Batterie, Panne, Constructeur, Accident*, etc. A titre d'exemple, l'attribut *prixConseillé* est mis dans la vue *CommercialVoiture* associée au commercial car les autres acteurs n'ont pas besoin d'avoir accès à cette information. La méthode *modifierInfos* de la classe *Voiture* est masquée au client car ce dernier ne doit pas modifier les informations d'une voiture.

Les classes *Moteur, BoiteAVitesses, Signalisation, Batterie, Carrosserie* et *Panne* sont reliées à la vue *MaintenicienVoiture* et non pas à la vue *ClientVoiture* ou à la vue *CommercialVoiture*, car les informations et les services de ces classes ne sont pas dédiés au client ni au commercial. Par contre la classe *Constructeur* est reliée à la base ; en effet, tous les acteurs ont droit de connaître les informations concernant le constructeur d'une voiture. La classe *PropositionDAchat* est reliée aux vues *ClientVoiture* et *CommercialVoiture* et pas à la vue *MaintenicienVoiture*, car les propositions de prix sont à la fois gérées par le client et le commercial et pas par le maintenicien.

Nous remarquons sur la figure 33 que les classes *Constructeur* et *Accident* sont stéréotypées par « *multiViewsClass* ». En effet, pour des raisons de lisibilités des diagrammes de classes VUML, nous avons introduit le stéréotype « *multiViewsClass* » qui permet de représenter des classes multivues non éclatées (iconifiées).

II.3.3. Hiérarchisation des vues d'une classe multivues

VUML adopte le principe d'une association déterministe d'une vue à un acteur, moyennant la simplification de langage mentionnée en II.3.1. Cependant, il se peut qu'une vue soit une sous-vue d'une autre vue. Par exemple, dans le cas de la voiture, l'entretien est réalisé par un maintenicien qui peut être soit un mécanicien, soit un électricien ou un carrossier-tôlier. On peut donc considérer que ces trois *acteurs concrets* spécialisent l'*acteur abstrait* maintenicien (cf. figure 34). Les vues associées à ces acteurs seront donc des sous-vues de la vue dite *abstraite* associée au maintenicien.

Dans ce cas, il faut disposer d'un mécanisme permettant la prise en compte des sous-vues au sein d'une classe multivues. Cela revient à définir un mécanisme de hiérarchisation des vues. Pour ce faire, nous proposons d'utiliser des vues hiérarchiques (à l'instar de la hiérarchie des classes) en définissant une vue, qui peut être abstraite, contenant tout ce qui est en commun et en créant les autres vues comme des descendantes (par héritage) de cette vue. Cette solution souffre cependant d'un inconvénient car l'héritage ne permet de partager que la structure et non les données. Pour remédier à ce problème, il faut gérer les dépendances entre les vues (cf. section II.3.5).

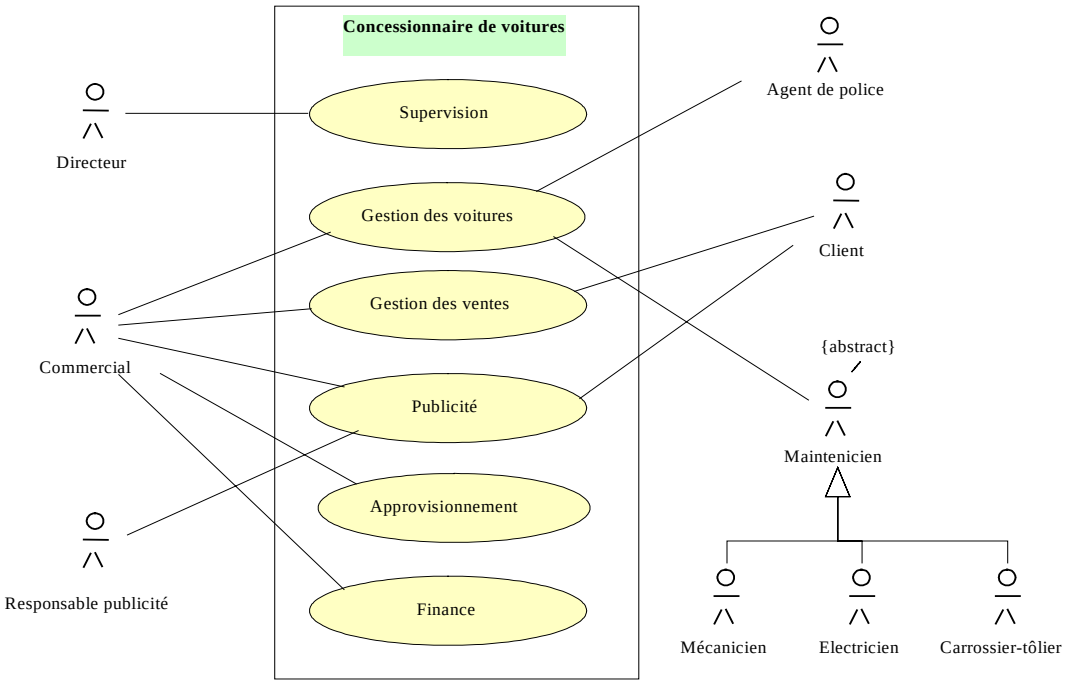

Figure 34 – *Diagramme des cas d'utilisation du système "concessionnaire de voitures" avec des acteurs spécialisant l'acteur abstrait Maintenicien*

La figure 35 donne un exemple abstrait de la structure d'une classe multivues ayant des sous-vues. Nous constatons qu'une vue ne peut hériter que d'une vue ou d'une vue abstraite (classe stéréotypée par « *abstractView* »). Une vue abstraite est une vue qui possède au mois une méthode abstraite, et qui ne peut pas être activée.

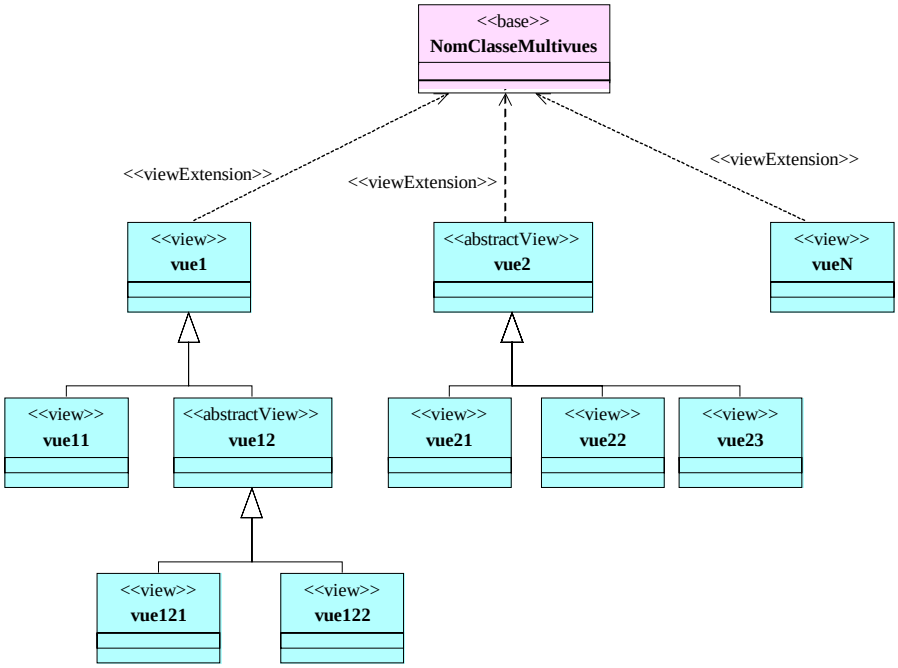

Figure 35 – *Exemple abstrait d'une classe multivues avec des sous-vues*

Afin d'illustrer la hiérarchisation des vues d'une classe multivues à travers l'exemple de « concessionnaire de voitures », nous allons prendre en considération les vues du mécanicien, de l'électricien et du carrossier-tôlier. La figure 36 ci-dessous représente le modèle VUML après l'intégration de ces sous-vues. Nous constatons que les classes composant la vue *MaintenicienVoiture* dans l'exemple de la figure 33 sont maintenant réparties sur les trois sous-vues : *MécanicienVoiture*, *ElectricienVoiture* et *Carrossier-TôlierVoiture*. Cette répartition est faite bien sûr en respectant la spécificité de la vue de chaque acteur. La vue abstraite *MaintenicienVoiture* contient les méthodes *enregistrerPanne()* et *réparerPanne()* qui sont abstraites et communes aux 3 points de vue de la maintenance.

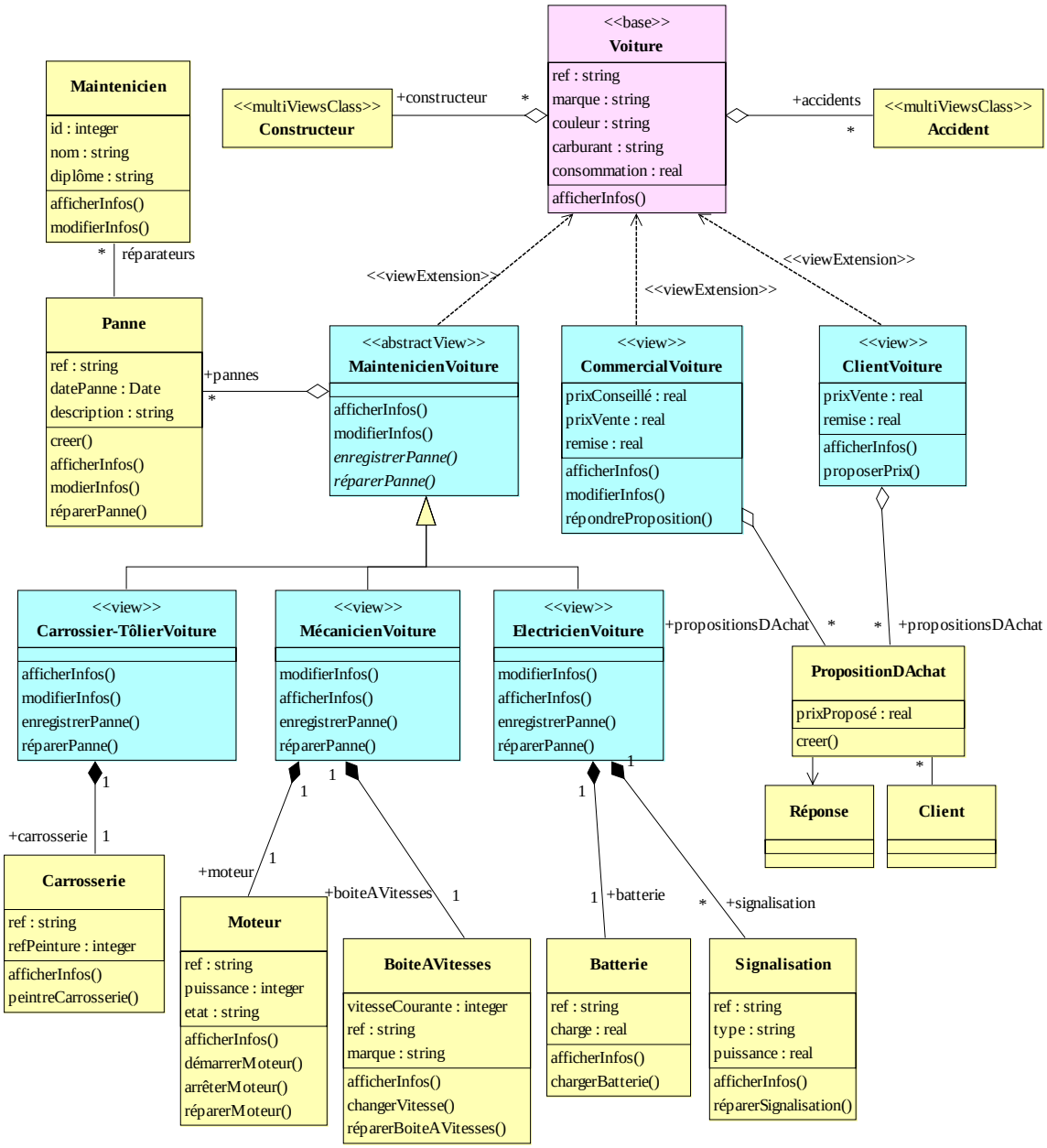

Figure 36 – *Modèle VUML du système "Concessionnaire de voitures" avec les sous-vues : MécanicienVoiture, ElectricienVoiture et Carrossier-TôlierVoiture*

II.3.4. Spécialisation d’une classe multivues

La relation de spécialisation n’est pas modifiée par l’introduction du concept de classe multivues. Elle correspond dans VUML à un héritage qui implique donc une relation de sous-typage et le fait que les caractéristiques d'une classe sont virtuellement copiées dans la sous-classe. Une classe multivues peut ainsi avoir des sous-classes qui sont automatiquement multivues. Le lien d’héritage est positionné entre les bases. Les vues de la classe parente deviennent des vues de la classe fille. Il est aussi possible de définir de nouvelles vues pour la classe fille ou de redéfinir une vue parente (Nassar et al., 2003a). Sur l'exemple de la voiture illustré ci-dessous (cf. figure 37) et sur lequel nous avons "iconifié" les vues *ElectricienVoiture* et *Carrossier-TôlierVoiture*, il est ainsi possible de définir une nouvelle classe *VoitureCourse* spécialisant la classe *Voiture*, en lui ajoutant une nouvelle vue correspondant au point de vue du commissaire chargé de vérifier la conformité de la voiture de course dans le contexte d'une compétition. La redéfinition de la vue *MécanicienVoiture* en *MécanicienVoitureCourse* se justifie par le fait qu'un mécanicien doit prendre en compte par exemple le volume du réservoir et le poids à vide de la voiture pour prévoir les arrêts de la voiture au stand.

II.3.5. Dépendances entre les vues

Les vues d’une classe multivues peuvent naturellement être dépendantes. Il est donc nécessaire de maintenir la cohérence interne d'une telle classe. Autrement dit, les modifications de valeurs d'attributs d'une vue peuvent avoir des répercussions sur les valeurs d'attributs d'autres vues. La gestion de ces répercussions fait généralement partie de la phase d’implémentation au sens où du code doit être introduit pour faire les mises à jour nécessaires.

Cependant, nous pensons que ces dépendances – caractéristiques du système modélisé – doivent être explicitées le plus tôt possible dans le développement, c'est-à-dire durant la phase de conception ; pour cela nous utilisons une relation de dépendance stéréotypée par « viewDependency », les notes d’UML et le langage OCL (Object Constraint Language).

La figure 38 ci-dessous présente un exemple abstrait d’une dépendance entre deux vues d’une classe multivues. La dépendance « viewDependency » entre les vues *vue1* et *vue2* indique que des données de la vue *vue1* (source de la dépendance) dépendent de certaines données de la vue *vue2* (cible de la dépendance). La relation entre ces données doit être décrite en OCL sur la note associée à cette dépendance. Il existe plusieurs types de dépendance entre les vues : dépendance d’inclusion de données, dépendance d’égalité de données, dépendance fonctionnelle, etc.

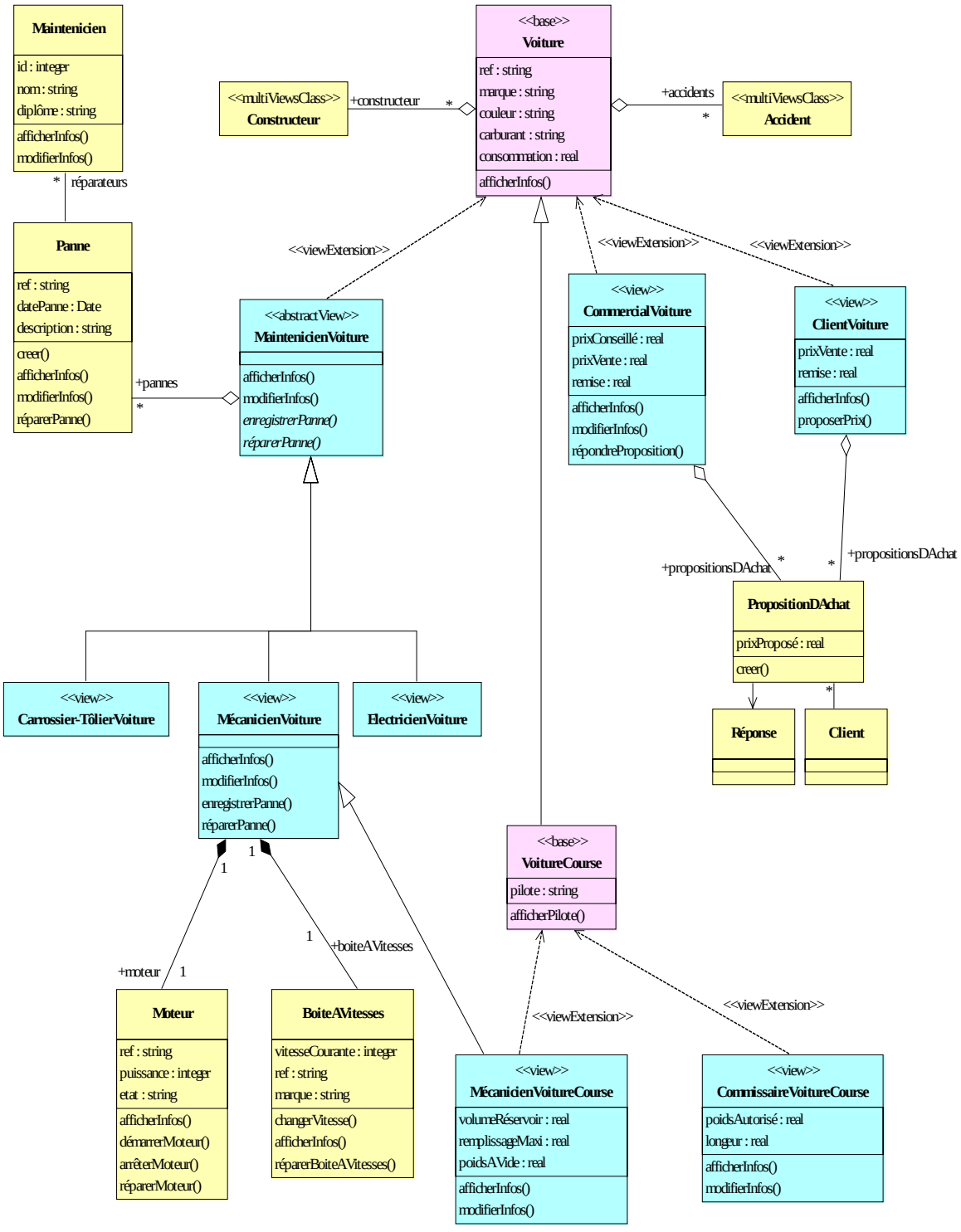

Figure 37 – *Illustration de la spécialisation d'une classe multivues*

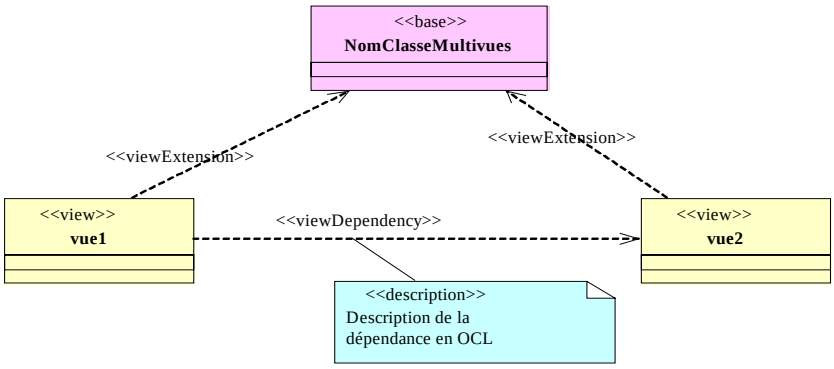

Figure 38 – *Illustration abstraite d'une dépendance entre deux vues*

Sur l'exemple de la figure 39 où certaines classes et vues sont "iconifiées" pour des raisons de lisibilité, nous avons mis en évidence 2 dépendances entre les vues *CommercialVoiture* et *ClientVoiture* de la classe multivues *Voiture*. Une première relation de dépendance « viewDependency » exprime que l'ensemble des propositions des prix reçues par le commercial relativement à une voiture inclut l'ensemble des propositions des prix effectuées par un client pour acheter cette voiture ; la seconde dépendance, toujours entre les vues *CommercialVoiture* et *ClientVoiture*, spécifie que le commercial et le client doivent utiliser les mêmes valeurs pour le prix de vente et la remise.

La figure 39 explicite aussi une troisième dépendance entre les vues *CommissaireVoitureCourse* et *MécanicienVoitureCourse* de la classe *VoitureCourse*. Cette dépendance exprime le fait que le taux de remplissage du réservoir (c'est-à-dire la quantité de carburant maximum à mettre) est fonction des attributs carburant (provenant de la classe parente), *volumeRéservoir* et *poidsAVide* appartenant à la vue initiale *MécanicienVoitureCourse*, et *poidsAutorisé* appartenant à la vue cible *CommissaireVoitureCourse*. La fonction *f* n'est pas donnée dans cet exemple simplifié.

II.3.6. Métamodèle de VUML

L'objectif du présent chapitre est de décrire notre approche de développement dont le noyau est un profil UML présenté plus loin dans ce chapitre. Les notions de profil et de métamodèle sont très proches et une utilisation combinée de celles-ci peut être bénéfique pour un projet (Peltier, 2002). Ceci favorise l'émergence de spécifications contenant à la fois la définition de leur profil et leur métamodèle MOF (Meta Object Facility). Dans cette optique, nous avons défini un métamodèle pour décrire la manière dont les concepts de notre profil s'intègrent dans le métamodèle UML. Ce métamodèle étend celui d'UML avec de nouveaux éléments de modélisation : *MultiViewsClass*, *GeneralView*, *Base*, *View*, *ViewExtension*, *ViewDependency*, *MultiViewsComponent* et *MVInterface*. L'extrait de ce métamodèle présenté sur la figure 40 montre qu'une classe multivues est composée d'une classe de base et d'un ensemble de vues. Ces dernières sont reliées à la base via une relation de dépendance particulière (extension) et peuvent être également reliées à d'autres vues par des relations de dépendances classiques (pour gérer la cohérence). Chaque vue est associée à un acteur unique. Le tableau de la figure 41 présente la liste des stéréotypes introduits (pour le concept de classe multivues). Pour chacun de ces stéréotypes, ce tableau décrit le nom, l'élément de modélisation associé et une description informelle de sa sémantique.

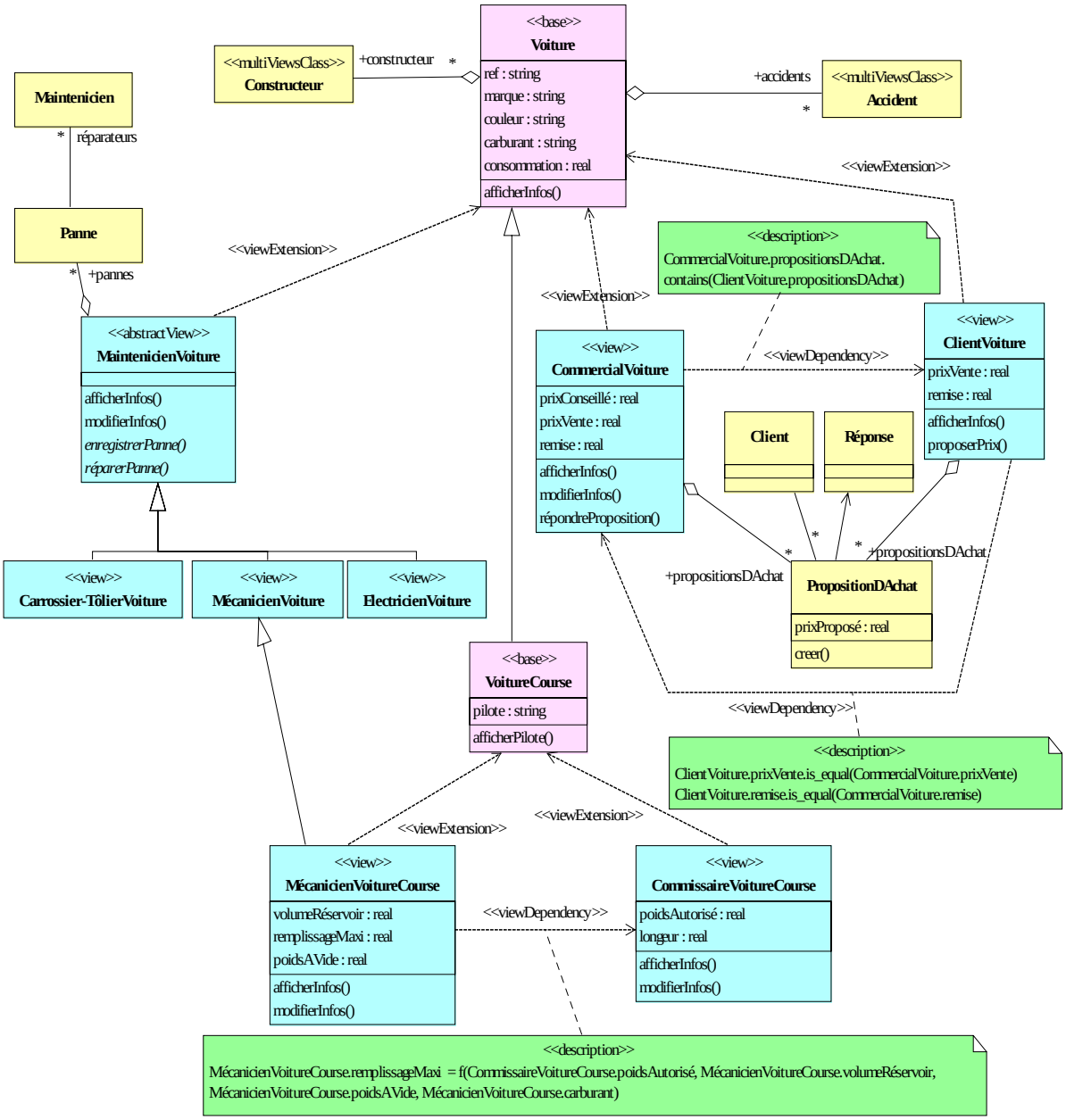

Figure 39 – *Illustration de dépendances entre vues*

II.4. De la classe multivues au composant multivues

L'introduction de la notion de classe *multivues* satisfait nos besoins d'analyse/conception jusqu'à l'établissement du diagramme de classes. Pour prendre en compte la transition vers le déploiement, l'évolution d'une architecture et la réutilisation d'éléments d'une application multivues, nous introduisons la notion de *composant multivues*. Un tel composant est la représentation au niveau du diagramme de composants d'une classe multivues.

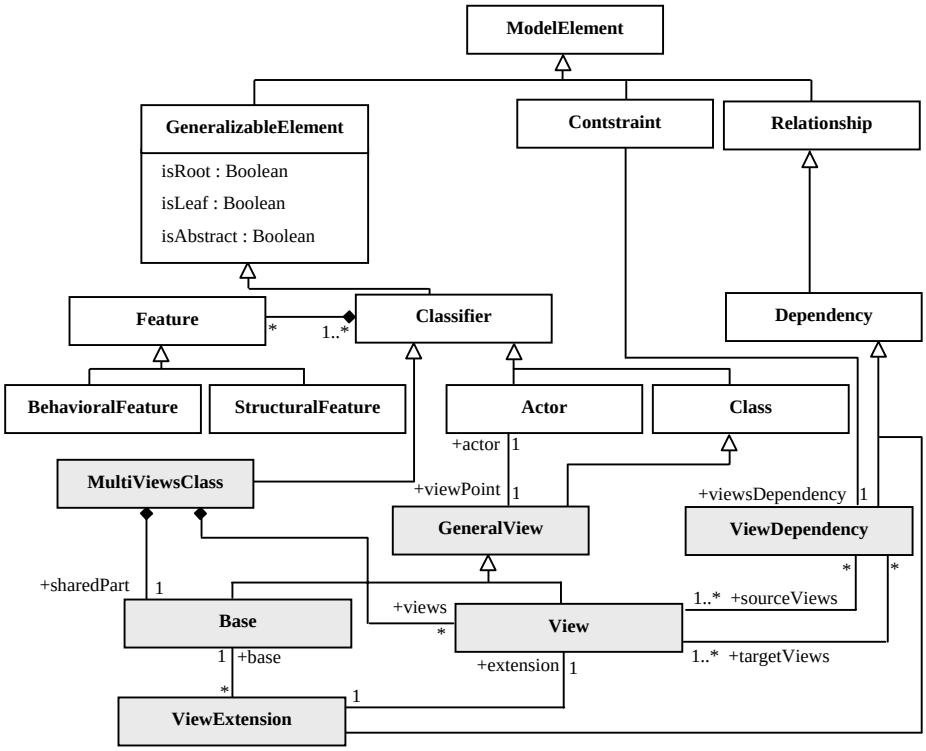

Figure 40 – *Fragment du métamodèle associé au profil VUML*

Stéréotype	Elément de modélisation	Sémantique informelle
base	Classe	Classe décrivant la base d'une classe multivues
view	Classe	Classe représentant une vue. Une telle classe doit être non multivues, ne peut hériter que de l'une des vues de sa classe multivues ou des vues de la classe parente de sa classe multivues, et ne peut pas être instanciée directement.
abstractView	Classe	Classe représentant une vue abstraite. Une telle vue ne peut pas être active.
viewExtension	Dépendance	Relation entre une vue et la base d'une classe multivues. Cette vue étend la base et peut redéfinir ses méthodes.
viewDependency	Dépendance	Relation entre les vues. Des informations de la vue source dépendent d'autres informations de la vue cible.
multiViewsClass	Classe	Classe représentant une classe multivues non éclatée (iconifiée)

Figure 41 – *Stéréotypes introduits dans le profil UML proposé*

Rappelons que les composants logiciels ont été introduits pour répondre aux besoins économiques. En effet, l'industrie du logiciel s'oriente vers un développement à base de composants. Il ne s'agit plus de concevoir "from scratch" des applications coûteuses mais de réaliser celles-ci par assemblage de composants logiciels existants. Ces derniers sont des unités logicielles analogues à des composants électroniques, mécaniques, etc. Il existe plusieurs modèles de composants : CCM, .NET, EJB, etc. Cependant, des recherches s'avèrent encore nécessaires pour disposer de composants logiciels fiables, sûrs, efficaces et réutilisables. Tenant compte de ces différentes exigences, cette section a pour but de présenter notre contribution qui consiste à combiner un développement par points de vue et les composants logiciels (Nassar et al. 2004b, El Asri et al. 2005a, El Asri et al. 2005b).

II.4.1. Concept de composant multivues

Un composant UML 2.0 est un module autonome doté d'un ensemble d'interfaces fournies et/ou requises. Une interface fournie est une interface offerte par le composant à son environnement. Cette interface peut être implémentée soit directement par le composant ou par l'une de ses parties, ou bien être le type d'un port fourni par le composant. Une interface requise est une interface que le composant requiert de la part d'autres composants pour réaliser ses services fournis. Une telle interface peut être utilisée par le composant ou ses parties, ou bien être le type d'un de ses ports requis (OMG, 2003b).

Dans le métamodèle d'UML 2.0, un composant est un sous-type de *Class* (cf. figure 42). Ainsi, un composant encapsule des attributs et des opérations, et peut participer à des relations d'association et de généralisation. Il est une abstraction d'un ensemble de *realizingClassifiers* qui réalisent son comportement. De plus, un composant peut avoir une structure interne et un ensemble de ports qui structurent des points d'interaction avec l'environnement extérieur.

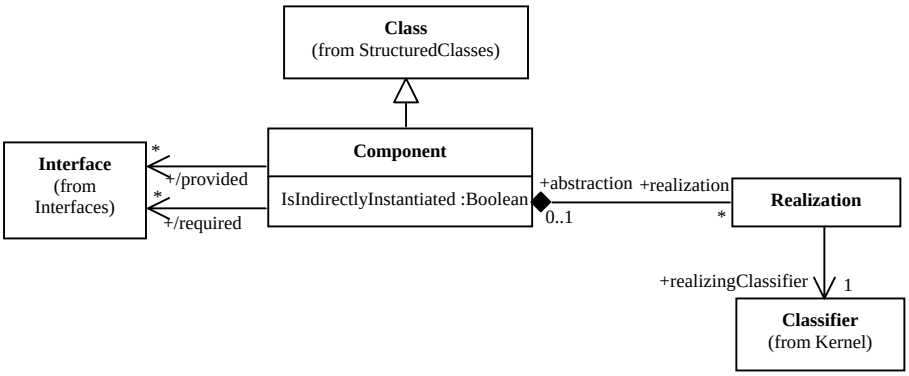

Figure 42 – *Métaclasse définissant le concept de composant en UML 2.0*

Par rapport aux composants UML "classiques", un composant multivues a la spécificité d'offrir des interfaces dont l'accessibilité et le comportement changent dynamiquement selon la vue correspondant au point de vue courant. En effet, un client accède à un composant selon un point de vue donné. Le client peut activer/désactiver des vues durant l'exécution. Par ailleurs, plusieurs clients peuvent accéder simultanément au même composant selon des points de vues différents (multi-utilisation). La figure 43 ci-dessous illustre la métaclasse *MultiViewsComponent* qui définit les éléments formant un composant multivues. Cette métaclasse montre qu'un composant multivues est une extension du composant UML 2.0.

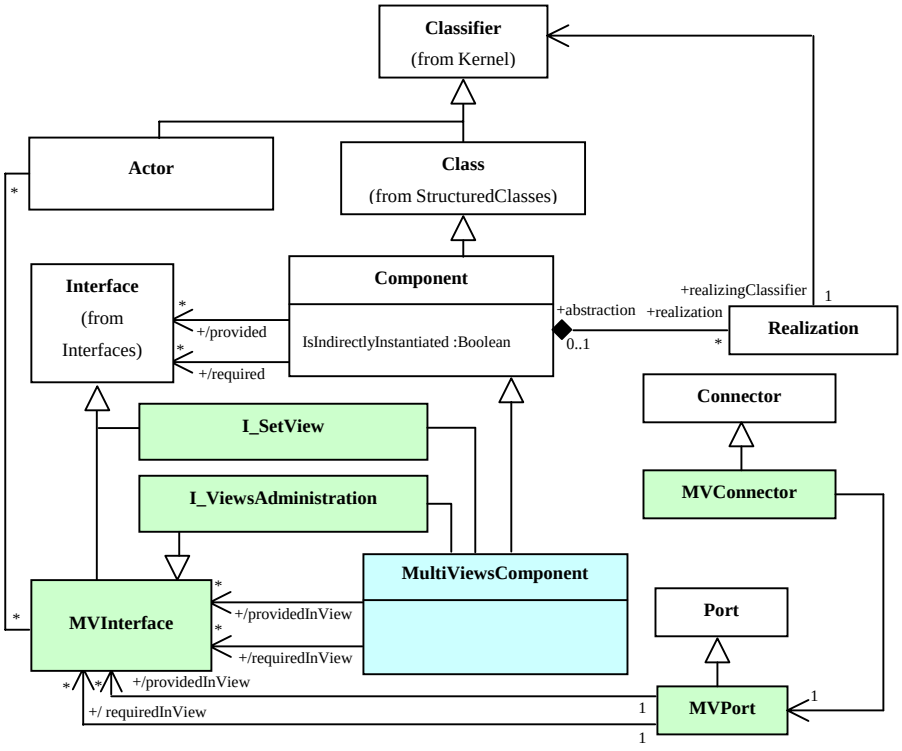

Figure 43 – *Métaclasse définissant le concept de composant multivues*

En plus des caractéristiques d'un composant UML 2.0 standard, un composant multivues est ainsi doté d'un ensemble *d'interfaces multivues* fournies et/ou requises qui décrivent le comportement du composant et son interaction avec l'environnement extérieur. Une *interface multivues* (MVInterface) est une interface qui définit, elle aussi, une relation de dépendance "utiliser/fournir" entre le composant et son environnement, mais le comportement des services offerts via cette interface change dynamiquement selon le point de vue actif. Un composant externe doit d'abord activer la vue adéquate avant de pouvoir interagir avec le composant multivues. Cette activation se fait à travers l'interface fournie (*I_SetView*) donnant accès à l'opération *setView().* Chaque composant multivues est doté d'une interface multivues fournie (*I_ViewsAdministration*) qui permet la gestion des vues (ajout/suppression, verrouillage/déverrouillage). Cette interface n'est accessible que par un administrateur (point de vue implicite). Un composant externe requérant cette interface ne peut y accéder que s'il active la vue *Administrateur* à l'exécution.

Un composant multivues doit fournir au moins une interface multivues. Le traitement des messages reçus dépend du point de vue du client. Bien entendu, un composant peut requérir des interfaces multivues sans être lui-même multivues. En effet, l'utilisation par un composant d'une interface multivues signifie que ce composant doit activer une vue donnée avant de solliciter les services d'un composant multivues. Les interfaces multivues doivent être organisées sous forme d'un seul *port multivues* portant le nom *MVPort*. La communication entre le composant et son environnement via un tel port est filtrée et routée vers les constituants internes (*parts*) correspondants selon la vue active. L'interaction via un port mutlivues est réalisée moyennant des *connecteurs multivues*. Ces derniers sont des extensions du connecteur UML classique. Ils permettent les assemblages de composants selon un point de vue déterminé, ce qui permet d'avoir une interaction conditionnée par la vue active (El Asri et al., 2005a).

La figure 44 ci-dessous explicite les interfaces (simples et multivues) pour le composant multivues *Voiture*. La notation utilisée est celle d'UML 2.0. A titre d'exemple, l'interface multivues fournie *Consultation* est implémentée par le composant. Elle est toujours accessible ; cependant, le comportement de la méthode qu'elle offre (*afficherInfos()*) change selon la vue active. L'interface multivues requise *Moteur*, accessible si la vue du mécanicien est active, permet de communiquer avec le composant *Moteur* fournissant cette même interface. L'interface multivues fournie *Panne*, accessible via le point de vue du mécanicien, de l'électricien et du carrossier, permet à ces acteurs de traiter les pannes mais chacun selon son point de vue (les méthodes *modifierInfos()*, *enregistrerPanne()*, et *réparerPanne()* se comportent différemment selon la vue active).

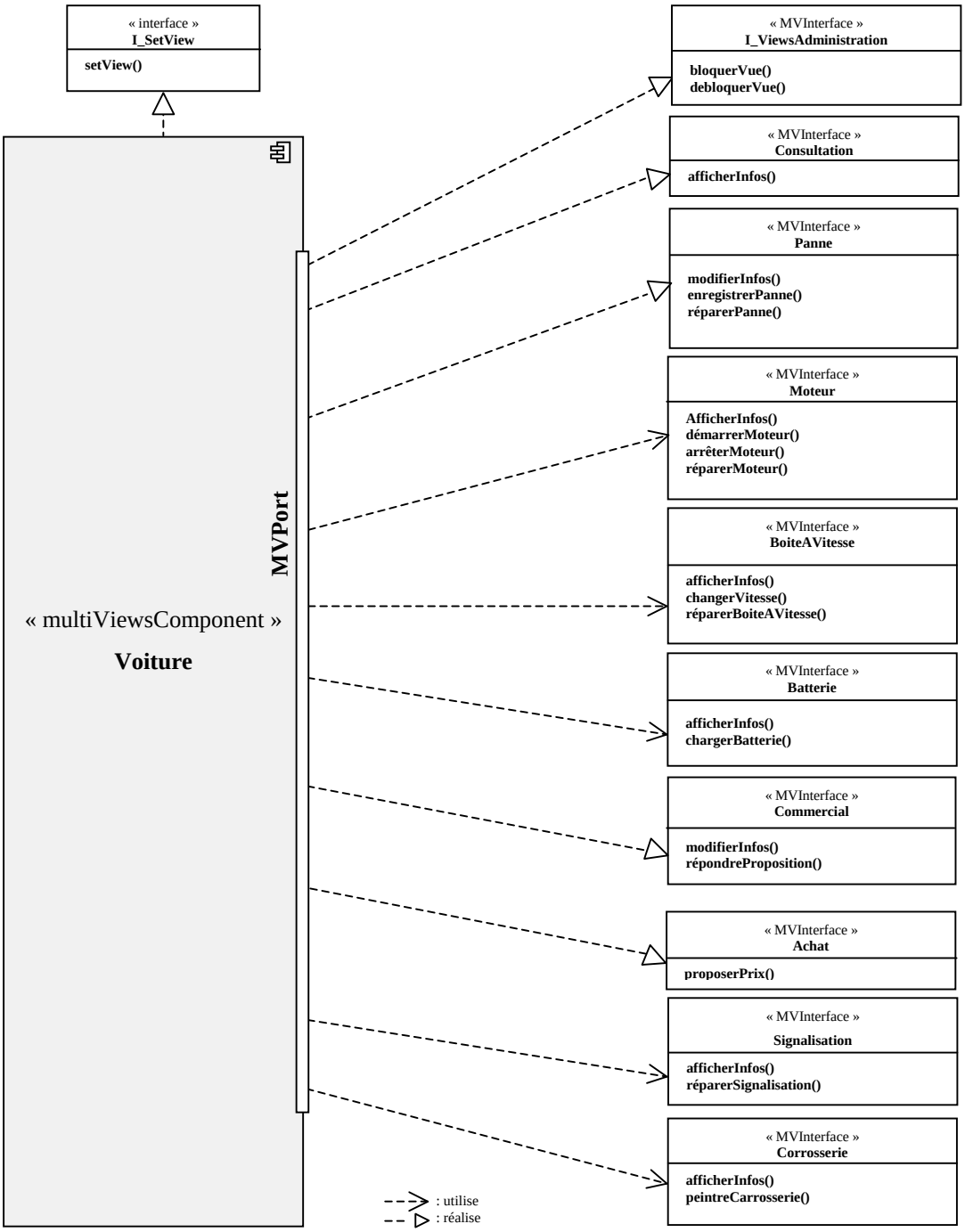

Figure 44 – *Représentation explicite des interfaces (simples et multivues) fournies et requises du composant multivues « Voiture »*

II.4.2. Principe d'assemblage de composants multivues

L'assemblage de composants permet de construire des systèmes avec des composants préexistants. Ceci est réalisé en établissant des connexions d'assemblage entre les composants. Cette approche, adoptée notamment dans UML 2.0, n'a pas été modifiée par l'introduction de la notion de composant multivues. Un composant multivues possède un ensemble d'interfaces (multivues et/ou simples) fournies et/ou requises. Ces interfaces forment la base pour l'assemblage de ces composants.

La figure 45 illustre un exemple d'assemblage des composants multivues *Voiture*, *Moteur*, *BoiteAVitesses*, *Batterie*, *Signalisation*, *Carrosserie*, *ServiceCommercial*, *Clientèle*, *ServiceMaintenance* et du composant classique *Web*. Le composant multivues *Clientèle* est connecté au composant multivues *Voiture* via l'interface multivues *Achat*. Cette interface nécessite l'activation de la vue *Client* sur le composant *Voiture* avant d'appeler une des méthodes fournies à travers cette interface. Le composant *Web* utilise l'interface simple I_*SetView* et l'interface multivues *Consultation* fournies par le composant *Voiture*. Le service offert à travers l'interface multivues *Consultation* dépend de la vue activée via l'interface I_*SetView*.

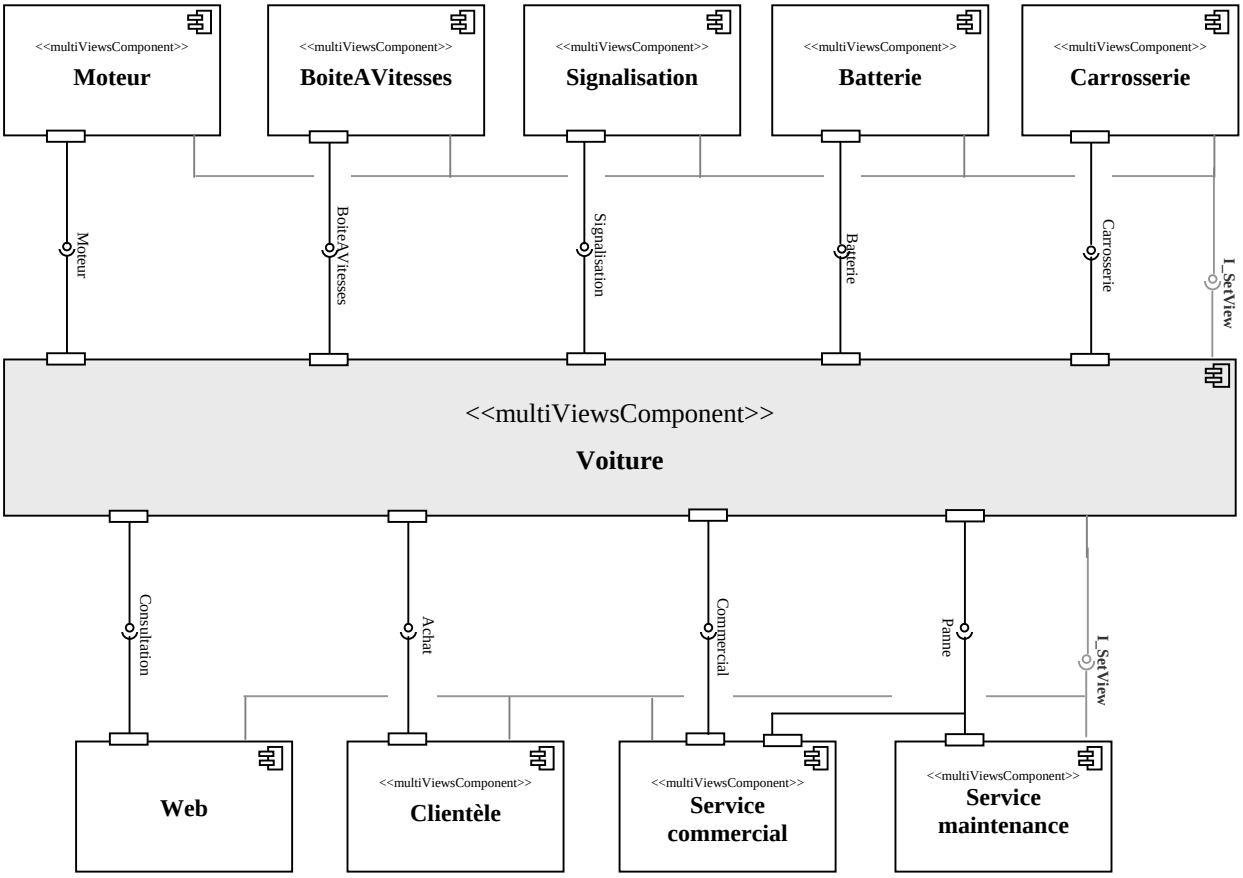

Figure 45 – *Exemple d'assemblage de composants multivues*

II.5. Eléments sur l'aspect dynamique de VUML

Jusqu'ici ce chapitre n'a présenté que les aspects statiques de l'approche VUML. Cependant, les aspects dynamiques sont aussi très importants à considérer. Ils concernent les diagrammes dynamiques

et l'implémentation. Les diagrammes dynamiques VUML qui dépendent d'un seul point de vue (tels que les diagrammes de séquence et les diagrammes de collaboration) sont similaires à leurs homologues dans UML. Par contre les diagrammes dynamiques VUML qui dépendent de plusieurs point de vue (tels que les diagrammes d'activité et les diagrammes d'état-transition) sont modifiés par rapport aux même genres de diagrammes UML. Mais l'étude de ces diagrammes dynamiques ne fait pas partie de cette thèse. Par conséquent, dans cette section, nous décrivons seulement les aspects concernant l'implémentation. Pour des raisons de simplification, nous ne considérons que les systèmes mono-utilisateur. Les systèmes multi-utilisateurs nécessitent des mécanismes de synchronisation qui sortent du cadre de la présente thèse.

II.5.1. Structure d'un objet multivues

Un objet multivues est une instance d'une classe multivues. Comme les objets classiques, un objet multivues a un état et un comportement. Il est composé d'un *objet base* et d'un ensemble d'*objets vues* (dits tout simplement vues par la suite). La gestion de l'ensemble de ces vues est assurée par un *gestionnaire de vues* qui se charge en particulier de l'activation/désactivation des vues (cf. figure 46). Ce gestionnaire communique avec l'extérieur moyennant deux interfaces qui lui permettent de traiter un certain nombre de demandes concernant la gestion des vues (activer/désactiver une vue, bloquer/débloquer une vue, récupérer la vue active).

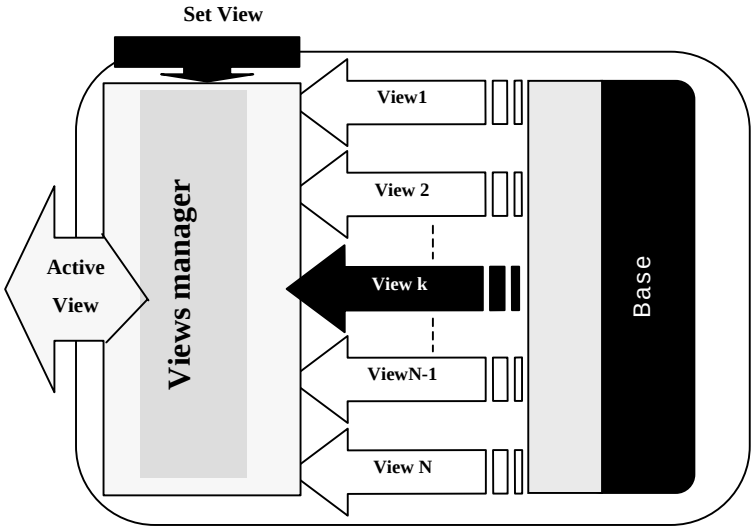

Figure 46 – *Structure d'un objet multivues*

II.5.2. Gestion des vues

Au moment de l'exécution, une vue d'un objet multivues est active, c'est-à-dire accessible, si elle est associée au point de vue de l'utilisateur courant. L'activation des vues est faite dynamiquement selon le point de vue courant. Plus précisément, nous distinguons 5 opérations sur les vues :

- création d'une vue
- activation d'une vue
- désactivation d'une vue
- blocage d'une vue
- déblocage d'une vue

L'objet multivues offre aussi un service qui permet d'administrer dynamiquement la liste des vues susceptibles d'être activées. Cette administration est une fonctionnalité supportée par tous les objets multivues et elle est accessible moyennant la vue *Admin.* ; en effet, chaque objet multivues possède cette vue (générée automatiquement). L'intérêt d'une telle fonctionnalité est d'avoir la possibilité de bloquer/débloquer dynamiquement l'activation de certaines vues. Ceci montre qu'il y a une gestion des droits d'accès à 2 niveaux : le premier niveau est réalisé au moment de la conception ; c'est-à-dire qu'une vue n'a pas le droit d'accéder aux ressources d'une autre vue. Le deuxième niveau se fait à l'exécution ; en effet, c'est un utilisateur qui a le privilège d'administrateur qui impose que certaines vues sont bloquées (lors de l'exécution). Cette vue d'administration est utilisée aussi pour initialiser les différents objets vues (cf. section IV.2.2.2.1.1.1).

II.5.3. Propagation de la vue active

L'activation d'une vue sur un objet concerne tous les objets multivues ayant des relations avec cet objet. A titre d'exemple, considérons une classe multivues *MVC1* reliée à une autre classe multivues *MVC2* via une relation d'association, d'agrégation ou de composition. Au moment de l'exécution, la vue active d'une instance de la classe *MVC1* doit être propagée à une ou à plusieurs instances de la classe *MVC2* (dépendant de la cardinalité de la relation entre *MVC1* et *MVC2*), et ainsi récursivement à tous les objets multivues liés. Si la vue active de *MVC1* n'a pas son correspondant dans *MVC2*, la propagation s'arrête.

Considérons le modèle VUML de la figure 47 ; l'activation de la vue *commercial* sur la *Voiture* est propagée au *Constructeur* et à chacun des objets de la liste *accidents*, tandis que l'activation de la vue *client* sur cette voiture ne peut pas se propager ainsi car le client n'a pas de vue sur la classe multivues *Constructeur* ni sur la classe multivues *Accident*.

II.5.4. Traitement des messages

L'appel d'une méthode existante sur un objet multivues se fait de la manière suivante : la méthode est cherchée tout d'abord dans la vue active ; si elle n'y est pas, elle est cherchée dans la base de l'objet multivues. Si cette méthode n'est pas trouvée et appartient à une autre vue, une exception est levée pour informer l'appelant que la méthode invoquée n'est pas accessible selon la vue courant. Ce traitement est réalisé en utilisant un mécanisme d'aiguillage d'appels qui redirige les messages vers les vues offrant les comportements demandés (cf. Implémentation - section VI.2.2.2, chapitre VI).

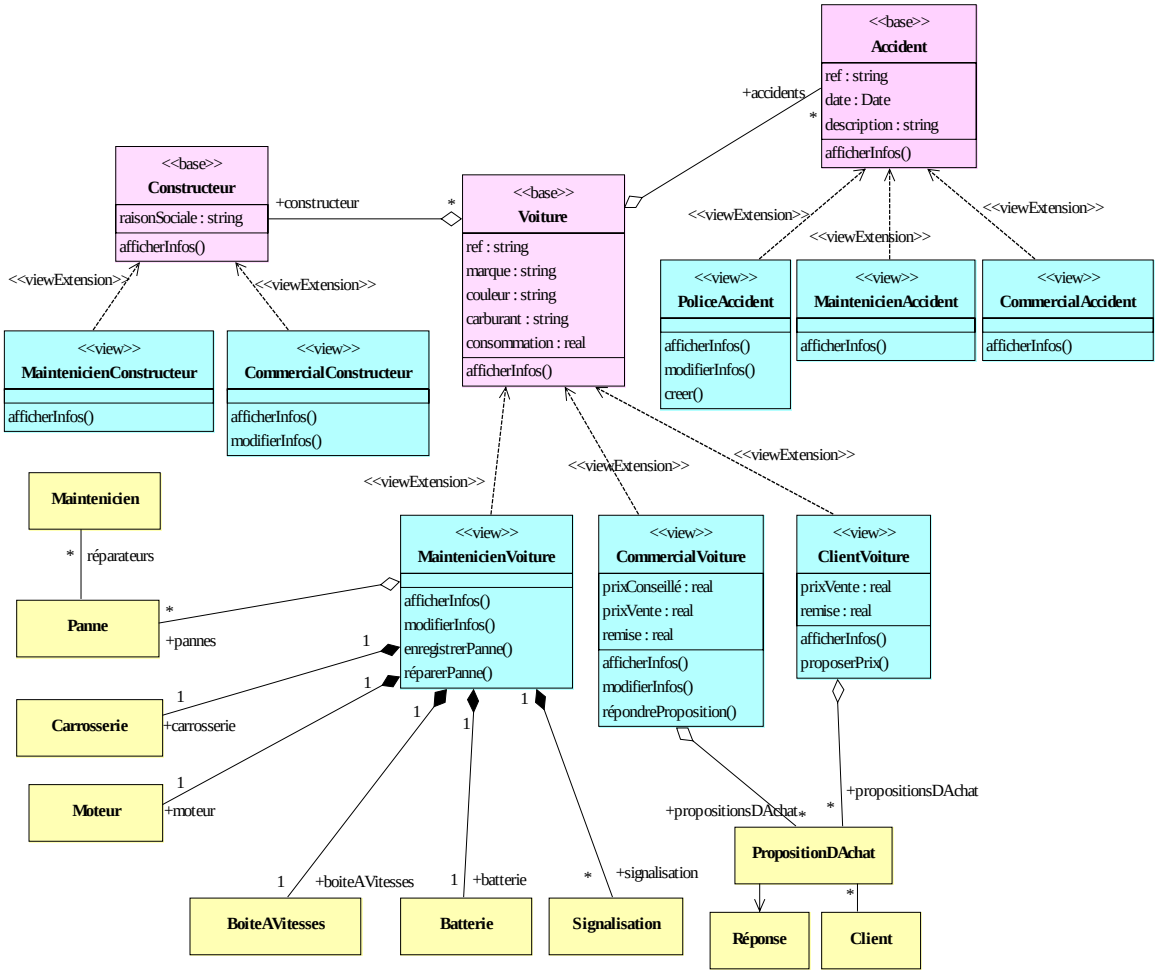

Figure 47 – *Exemple de modèle VUML avec trois classes multivues*

II.5.5. Génération de code multi-cibles dans VUML

Pour que notre approche soit convaincante et facile à utiliser, il ne faut pas qu'elle pénalise le programmeur objet "classique". C'est ce que nous avons recherché en ciblant d'une part les langages à objet du marché, et d'autre part en proposant une traduction automatique d'une classe multivues en un modèle d'implémentation. Nous donnons ci-dessous à cet effet quelques éléments concernant un patron d'implémentation générique qui permet de produire, à partir d'une classe multivues, un code objet multi-cibles (Java, C++, Eiffel, …). Le lecteur pourra trouver une description détaillée de la technique employée dans (Nassar et al., 2002, Nassar et al. 2004a).

La figure 48 ci-après illustre la structure statique du patron proposé. Ce patron vise à utiliser la délégation et le polymorphisme pour gérer les droits d'accès aux différentes vues, et le changement dynamique de vue. Il est inspiré du patron *stratégie* (Gamma et al., 1995).

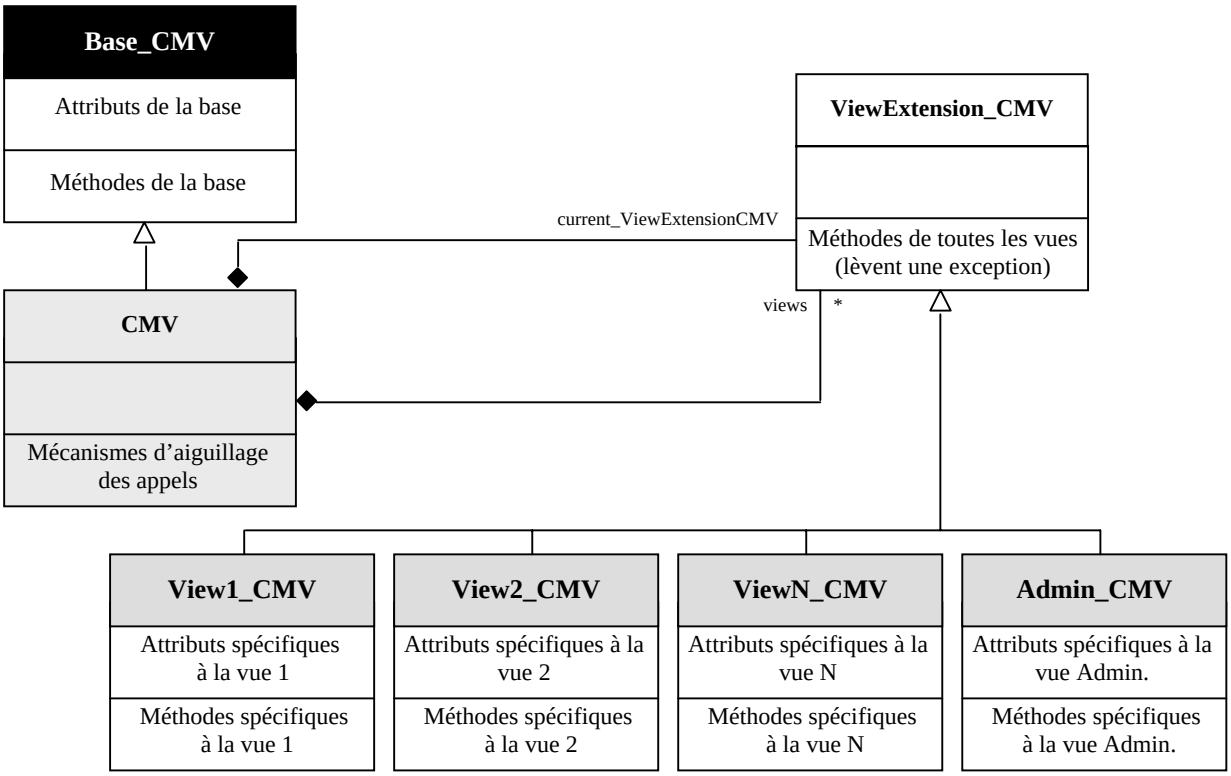

Figure 48 – *Patron d'implémentation pour la génération de code objet multi-cibles (1ère version)*

Considérons une classe multivues appelé *CMV* (cf. figure 48). L'ensemble de classes générées (dans n'importe quel langage orienté objet) comprend une classe appelée *base_CMV* contenant les attributs et les méthodes de la base de la classe multivues *CMV*, une classe *CMV* (classe qui représente la classe multivues dans le code généré) contenant les mécanismes d'aiguillage des appels vers la vue active. La classe *CMV* générée est reliée par composition à la classe *ViewExtension_CMV* contenant les méthodes provenant de toutes les vues de la classe multivues *CMV*, avec un comportement qui se réduit à lever une exception ("accès interdit"). Le lien entre la classe *CMV* et la classe *ViewExtension_CMV* se traduit par la génération de l'attribut *current_ViewExtensionCMV* au niveau de la classe *CMV*. La classe CMV est reliée aussi par composition à une liste de classes correspondant aux vues. En effet, la classe *ViewExtension_CMV* possède un ensemble de classes descendantes *View1_CMV*, *View2_CMV*,... contenant les attributs et les méthodes spécifiques des vues *View1*, *View2*, ... Ceci permet d'affecter un objet de type *Viewi_CMV* à un objet de type *ViewExtension_CMV* (Polymorphisme). Ainsi, il est possible d'activer et de désactiver les vues en changeant le type dynamique de l'attribut c*urrent_ViewExtensionCMV*. D'autre part, pour que l'utilisation ou la redéfinition des méthodes de la base et l'utilisation de ses attributs dans les vues soient possibles, la classe *ViewExtension_CMV* (et par héritage ses classes descendantes) possède une référence vers la classe *CMV*. Pour ce faire, cette classe (par exemple en Java) est dotée d'un constructeur à un seul argument de type *Object*, et la création de l'attribut c*urrent_ViewExtensionCMV* se fait en spécifiant la référence de la classe *CMV* (*this*) comme paramètre.

Le patron proposé ne dépend d'aucun langage cible particulier, il a été expérimenté avec Java pour réaliser un prototype d'un système d'enseignement à distance (Nassar et al., 2002). Des détails techniques concernant ce patron sont donnés dans la section IV.2.2.2 du chapitre IV.

Cependant, le patron proposé ne supporte pas certains mécanismes de VUML : la propagation de la vue active et la multi-utilisation (accès concurrents). Pour cela, nous avons proposé une deuxième version de ce patron qui combine le patron *Rôle* pour implémenter la délégation et la technique de la poignée, et le patron *Stratégie* pour l'implémentation des vues avec la surcharge des méthodes (Gamma et al. 1995, Coad 1992). Nous ne donnons que les éléments clés permettant de comprendre ce patron, en laissant de côté les détails de l'implémentation. Une description détaillée de la technique employée peut être trouvée dans (Crégut et al., 2005).

La figure 49 ci-après illustre la structure statique de la deuxième version du patron. Une classe multivues *CMV* est traduite par l'ensemble de classes suivant : une classe de même nom *CMV* reliée par composition à une classe *Base_CMV* (représentant la base), et à une liste de classes correspondant aux vues. Les vues sont décrites par des sous-classes de *ViewExtension_CMV*. La vue active est soit une instance de *ViewExtension_CMV*, soit une instance de la classe décrivant une vue (*Viewk_CMV*). Les méthodes sont appliquées sur l'instance de la classe *CMV* qui contient toutes les méthodes définies dans la classe multivues. La sélection de la méthode à exécuter s'appuie sur le polymorphisme de *ViewExtension_CMV*.

Considérons une méthode *m(args)* de la classe multivues *CMV* ayant T comme type de retour. La définition de la méthode *m()* engendrée dans la classe *CMV* se contente de rediriger l'appel vers la vue courante. La définition de la méthode *m()* dans la classe *ViewExtension_CMV* dépend du lieu de définition de *m()*. Si *m()* est dans la base, l'appel engendré passe par une indirection sur elle. Si *m()* n'appartient pas à la base, le comportement de *m()* est réduit à lever une exception interdisant l'appel.

La différence majeure par rapport à la première version du patron est le fait que pour chaque méthode *m()* de la classe multivues on ajoute un paramètre (le premier) de type *Utilisateur* (*Utilisateur* est une classe système qui possède un attribut (*rôle*) de type *String* pour stocker la vue active et un *accesseur* et un *setteur* pour accéder à cet attribut). Ce paramètre permet d'indiquer la vue active lors de l'exécution d'une méthode. Cette manière de procéder va permettre la multi-utilisation et la propagation de la vue active. En effet, contrairement à la première version du ptraon où la vue active est stockée dans l'attribut *current_ViewExtensionCMV*, la vue active n'est plus liée à la classe multivues mais transmise comme paramètre de la méthode invoquée.

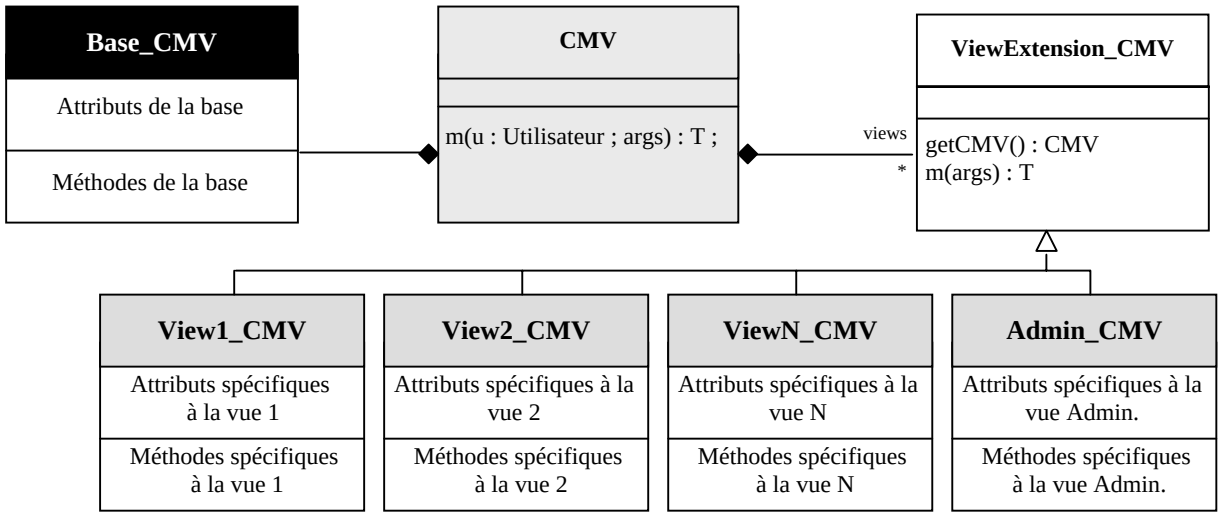

Figure 49 – *Patron d'implémentation pour la génération de code objet multi-cibles (2ème version)*

II.6. Le profil VUML

La notion de Profil UML a été introduite dans le standard UML 1.3 comme un moyen permettant la structuration des extensions UML (stéréotypes, contraintes et valeurs étiquetées). En effet, UML est un langage de modélisation à destination d'un grand nombre de domaines d'application. Cependant, chaque domaine a des spécificités, des besoins particuliers, que UML peut supporter par le biais de ses extensions, regroupées en « Profils UML ».

Dans ce contexte, l'objectif de cette section est de structurer notre approche VUML – qui propose une extension du métamodèle UML – sous forme d'un profil UML. Nous commençons par présenter la notion de profil, en prenant en compte les dernières évolutions de cette notion.

II.6.1. Notion de profil dans UML

Par rapport au formalisme UML standard, les développeurs souhaitent souvent rajouter des caractéristiques supplémentaires pour tenir compte de la spécificité de leur domaine d'application. Afin de satisfaire ce besoin, UML est doté d'un mécanisme d'extensibilité fondé sur les stéréotypes, les contraintes et les valeurs étiquetées. Un tel mécanisme permet de personnaliser le métamodèle UML pour qu'il prenne en considération les besoins de modélisation spécifiques. Le résultat de cette personnalisation est un profil UML. Plusieurs profils UML ont été standardisés ou sont en cours de standardisation, tels que : SPEM (modélisation des procédés logiciels), EDOC (modélisation des applications distribuées), Scheduling, Performance and Time (modélisation des systèmes embarqués), UML pour CORBA, UML pour les EJB, etc. (cf. OMG-site, 2004).

Un profil UML peut être décrit simplement comme un ensemble d'éléments de modélisation ajoutés au métamodèle UML (OMG-site, 2004). On peut cependant lui ajouter la notion de règles comme préconisé par SOFTEAM (SofTeam, 1999). Ces règles permettent de décrire et d'automatiser un savoir-faire sur UML. De cette façon, les profils UML constituent un moyen efficace pour spécifier et guider le processus de développement UML. Durant le développement, à chaque phase, les profils permettent d'exprimer comment utiliser UML, quels sont les produits de développement attendus, et les règles que le modèle doit respecter. En reprenant cette approche (SofTeam, 1999), un profil UML peut être décrit par :

- les éléments UML utilisés (éléments d'UML pertinents pour un domaine donné),
- les extensions UML ajoutées (stéréotypes, contraintes, valeurs étiquetées),
- les règles de validation (règles vérifiant des critères de cohérence sur un modèle pour un profil donné),
- les règles de présentation (les diagrammes UML doivent présenter certaines informations et en cacher d'autres),
- les règles de transformation (règles de génération de code et patrons permettant d'assister ou d'automatiser le développement).

La figure 50 ci-dessous présente un exemple d'une description d'un profil UML d'analyse selon l'approche SofTeam.

Eléments UML utilisés	Package, Classe, Use Case
Extensions UML ajoutées	Steréotype : « *Business_Object* » (classe). Tagged Value : {analysis}
Règles de Validation	Métriques : 10 classes max par package ; 10 opérations max par classe. Tous les acteurs doivent coopérer avec au moins un Use Case. Détection des objets sans classes, des messages sans opérations.
Règles de présentation	Diagrammes de classes et de Use Case. Seules les opérations publiques sont affichées. Visualisation particulière des classes *Business_Object*.
Règles de transformation	Plan type de documentation. Complémentation automatique des diagrammes

Figure 50 – *Exemple de contenu d'un Profil UML d'analyse (tiré de SofTeam 1999)*

II.6.2. Présentation du profil VUML

Dans cette section, nous présentons le profil UML proposé pour supporter l'approche VUML. Le formalisme de description utilisé est celui de SOFTEAM (SofTeam 1999). La description est faite en 5 rubriques (cf. figure 51). Les règles de validation portent sur la classe multivues et le composant multivues. Une description détaillée de ces règles est présentée dans le chapitre III. Les notations boîte blanche et boîte noire mentionnées en règle de présentation correspondent aux deux modes de représentation d'un composant illustrées respectivement sur les figures 44 et 45.

Eléments UML utilisés	Classe, Relation de dépendance, Note, Composant, Interface, Code OCL
Extensions UML ajoutées	Stéréotypes : base, view, abstractView, viewExtension, viewDependency, multiViewsClass, multiViewsComponent, MVInterface
Règles de Validation	- Une « base » a, au moins, une relation « viewExtension », ou doit être un descendant direct d'une « base » ou d'un « multiViewsClass ».

	- Un descendant direct de « base » est soit une « base » soit un « multiViewsClass ». - Un élément « view » a, au plus, un parent. - Un élément « view » ne peut hériter que de « view » ou de « abstractView ». - Un descendant direct d'un « view » est soit un « view » soit un « abstractView ». - Un « view » doit être source d'une seule relation « viewExtension », ou être descendant d'un et un seul « view » ou d'un et un seul « abstractView ». - Si un « view » est relié par une relation « viewExtension » à une base B1, et hérite d'un « view » ou « abstractView » d'une base B2, alors B1 est un descendant de B2. - Un descendant direct de « multiViewsClass » est soit un « multiViewsClass » soit une « base ». - La dépendance « viewExtension » a pour source un « view » ou « abstractView » et a pour destination une « base ». - La dépendance « viewDependency » a pour source un « view » ou « abstractView » et a pour destination un « view » ou « abstractView ». - Si la source d'une dépendance « viewDependency » a comme base B1 et la destination de cette même dépendance a comme base B2, alors soit B2=B1 soit B1 est un descendant de B2. - Un « multiViewsComponent » doit avoir une seule interface fournie I_SetView. - Un « multiViewsComponent » doit avoir au moins une interface multivues. - Un « MVInterface » ne peut être qu'une interface d'un « multiViewsComponent ». - Un « MVInterface » est associé à au moins un acteur. - Soit R une association entre deux bases B1 et B2. R est valide si et seulement s'il existe un acteur A associé à une vue de B1 et associé à une vue de B2. - Soit R une association entre une base B et une vue V d'une autre base et soit A l'acteur associé à la vue V. R est valide si et seulement si l'acteur A est associé à une vue de B. - Toute association entre des vues concrètes ou abstraites est interdite. - Si une vue est reliée par une association avec une classe C (qui n'est pas une vue) alors cette association doit être uniquement navigable dans le sens vue vers C. - Une vue ne peut jamais jouer le rôle d'agrégé dans des relations d'agrégation. - Si une vue est reliée par une relation d'agrégation à une classe C (qui n'est pas une vue), alors la classe C doit être l'agrégé de cette relation, et cette agrégation doit être uniquement navigable dans le sens vue vers C. - Une vue ne peut jamais être un composant dans une relation de composition. - Soit R une relation de composition reliant une base B1 (composite) à une base B2 (composant), et soient V1, V2, …, Vn les vues de la base B2 associées, respectivement, aux acteurs A1,

Règles de présentation	Diagrammes de classes, Diagrammes de composants, Notation boîte noire ou Notation boîte blanche
Règles de transformation	Génération automatique du code via un patron spécifique (cf. section II.5.5)

Figure 51 – *Profil UML support de l'approche VUML (résumé*)

II.7. Conclusion

Le travail présenté dans ce chapitre s'inscrit dans le cadre des travaux de recherche que notre équipe mène sur l'élaboration d'une méthodologie d'analyse/conception par points de vue, appelée VUML (View based Unified Modeling Language). Nous avons tout d'abord introduit nos définitions concernant les notions de vue et de point de vue. Ensuite, nous avons présenté la notion de classe multivues – élément clé de notre approche – qui permet d'intégrer les concepts de vue et point de vue dans l'analyse/conception d'un système. Une classe multivues est une entité de modélisation "flexible" qui permet de décrire l'information en fonction des points de vue des acteurs concernés. Chaque classe multivues est statiquement composée d'une base et d'un ensemble de vues étendant cette base. Une classe multivues peut être spécialisée ; la classe résultante est aussi multivues, sa base héritant de celle de la classe parente. Nous préconisons la mise en évidence des dépendances entre les vues d'une classe multivues pendant la phase de conception à travers des déclarations de dépendances (explicitées en OCL).

Dans ce chapitre, nous avons aussi présenté la notion de composant multivues qui permet de représenter une classe multivues au niveau d'un diagramme de composants. Structurellement, un composant multivues étend le composant UML 2.0 en proposant des interfaces multivues (requises et/ou fournies). Une interface multivues est un cas particulier d'interface, accessible uniquement si la vue associée est active. Les interfaces multivues sont structurées en ports complexes qui gèrent l'interaction entre composants et permet leur assemblage.

Notre soucis étant de faciliter la transition vers le codage, nous avons aussi défini un patron d'implémentation générique qui décrit la manière de passer d'une modélisation VUML à du code objet standard (testé avec Java pour l'instant).

L'ensemble des stéréotypes introduits pour spécialiser UML est regroupé sous forme d'un profil UML. Ce profil a été implanté en utilisant l'outil *Objecteering/UML Profile Builder* (cf. chapitre IV).

Cependant l'approche VUML possède quelques limites notamment quand on ajoute une vue à une classe multivues. En fait, ce genre d'opération peut avoir des effets sur le contenu de la base de la classe multivues, car la base est sensée contenir les informations partagées par tous les acteurs ayant des vues sur la classe multivues. Ainsi, il se peut que certaines informations de la base ne doivent pas être accessibles à travers la nouvelle vue. Par conséquent il faut déplacer ces informations vers les vues et gérer éventuellement leur cohérence en utilisant des dépendances «viewDependency ». Ceci peut impliquer une évolution fastidieuse du système. Pour éviter ce problème, on peut réduire la base d'une classe multivues aux informations dont l'accessibilité reste intacte lors de l'évolution du système (base irréductible). Cette décision délicate peut amener à une base très petite voire vide, ce qui peut engendrer la prise en charge de plus de dépendances entre les vues.

Chapitre III

Sémantique de VUML

III.1. Introduction

Dans le chapitre II nous avons présenté notre approche VUML qui a pour objectif de permettre une modélisation basée sur les vues des utilisateurs. Nous avons vu que VUML offre une notation (extension d'UML) et une démarche permettant de mener le développement d'un système de l'analyse jusqu'à l'implémentation. L'extension d'UML proposée est regroupée sous forme d'un profil UML (cf. chapitre II, section II.6). L'objectif du présent chapitre est de décrire la sémantique associée à chaque élément de modélisation introduit dans ce profil. A l'instar d'UML, la sémantique statique de VUML est définie par le méta-modèle, les WFR (well-formedness rules) exprimées en langage formel OCL (Object Constraint Language) et des descriptions textuelles informelles. La sémantique dynamique de VUML est décrite par contre en langage naturel (Français).

Ce chapitre est organisé comme suit : nous donnons, tout d'abord, un aperçu sur la technique utilisée pour décrire la sémantique d'UML (section III.2), puis nous présentons, brièvement, le langage formel OCL (section III.3). La dernière section est consacrée à la présentation de la sémantique statique et de la sémantique dynamique de VUML.

III.2. Aperçu sur la technique de description de la sémantique d'UML

UML est un langage de modélisation qui fournit les fondements pour spécifier, construire, visualiser et décrire les artefacts d'un système logiciel. Afin d'assurer cela, UML se base sur une sémantique "précise" et sur une notation graphique dont la synatxe est à la fois simple, intuitive et expressive.

Pour faciliter la définition et la formalisation d'UML, les différents concepts d'UML sont modélisés eux-même en UML. Cette définition récursive, appelée métamodélisation, décrit de manière formelle les éléments de modélisation d'UML ainsi que la syntaxe et la sémantique de la notation qui permet de les manipuler. Le métamodèle devient, entre autres, un outil de vérification qui facilite l'identification des éventuelles incohérences, notamment par l'utilisation de règles de bonne modélisation[5], exprimées en langage OCL.

La sémantique d'UML comprend un aspect statique et un aspect dynamique. Afin de bien définir cette sémantique, une technique formelle est adoptée. Cette technique vise à améliorer la précision tout

[5] qui peut être une traduction de *well-formedness rules*

en maintenant la lisibilité. Elle décrit le méta-modèle UML en combinant une notation graphique, un langage naturel, et un langage formel. Le bénéfice de l'utilisation d'une technique formelle inclut :

-une amélioration de l'exactitude de la description d'UML,

-une réduction des ambiguïtés et des contradictions, et

-une augmentation de la lisibilité de la description d'UML.

Il est important de noter que UML n'est pas complètement spécifier d'une manière formelle. Ceci se justifie par le fait qu'une formalisation complète augmentera significativement la complexité du formalisme sans avoir de clair bénéfice (OMG, 2003a).

La sémantique dynamique d'UML est décrite à l'aide du langage naturel mais d'une manière précise de telle façon qu'elle soit facilement compréhensible. Actuellement, la sémantique dynamique n'est pas considérée comme essentielle pour le développement des outils ; cependant, ceci peut, probablement, changer dans le futur (OMG, 2003a).

Finalement, nous tenons à signaler que malgré les améliorations apportées par la dernière version du langage UML (UML 2.0) (OMG, 2003b), il ne possède pas encore une sémantique entièrement formelle. En effet, UML avec sa sémantique semi-formelle souffre toujours de problèmes d'ambiguïtés, d'imprécisions et même de contradictions. Une sémantique formelle permettrait d'avoir des outils pour vérifier et simuler l'exécution de modèles UML, générer du code à partir de ceux-ci, et aussi de tester la compatibilité entre les nouvelles extensions du langage et les concepts existants, etc.

III.3. Langage OCL (Object Constraint Language)

Les diagrammes UML ne permettent pas de décrire tous les aspects relevant de la spécification d'un système. D'où la nécessité d'avoir la possibilité de décrire des contraintes supplémentaires sur les objets d'un modèle. Ces contraintes qui sont des informations importantes ne peuvent pas être spécifiées à l'aide des éléments de modélisation définis par UML. Elles sont souvent exprimées en langage naturel ; cependant, l'utilisation d'un langage non formel peut introduire des ambiguïtés. Au contraire l'utilisation d'un langage formel supprime ces ambiguïtés en se basant sur une grammaire précise. Dans cette optique, l'OMG a adopté le langage OCL (Object Constraint Language) (OMG, 2003c) pour décrire une partie de la sémantique d'UML et comme complément des différents diagrammes UML permettant d'exprimer des contraintes (OMG, 2003b).

Initialement développé par IBM, OCL est un langage formel pour l'expression des contraintes. Il est simple à écrire et à comprendre, et raisonnablement puissant. Il représente un juste milieu entre langage naturel et langage mathématique. OCL permet ainsi de limiter les ambiguïtés, tout en restant accessible. Il est largement utilisé pour la description des méta–modèles dont celui d'UML. La grammaire complète d'OCL est donnée en annexe A.

Comme OCL est un langage déclaratif sans effet de bord, les contraintes OCL exprimées dans le modèle ne modifient pas les instances du modèle. OCL permet l'expression des contraintes suivantes :

- Les invariants au sein d'une classe ou d'un type : contraintes qui doivent toujours être vérifiées pour s'assurer du bon fonctionnement des instances de la classe ou du type concerné ;
- Contraintes au sein d'une opération : contraintes qui doivent toujours être vérifiées pour s'assurer de la bonne exécution de l'opération concernée ;

- Les pré- et les post-conditions d'opération : contraintes qui doivent être respectivement vérifiées avant et après l'exécution d'une opération ;
- Les gardes : contraintes sur la modification de l'état d'un objet ;
- Les expressions de navigation : contraintes pour représenter les chemins au sein de la structure de classes.

OCL souffre cependant d'un certain nombre de lacunes à savoir :

- Peu lisible pour des contraintes complexes ;
- Pas aussi rigoureux qu'un langage de spécification comme Z ou B (pas de preuves possibles) ;
- Puissance d'expression limitée.

III.4. Description de la sémantique de VUML

Dans cette section, nous décrivons la sémantique statique et la sémantique dynamique de VUML. Comme nous l'avons déjà dit, la définition de la sémantique statique sera faite par le méta-modèle, des règles de bonne modélisation (well-formedness rules) exprimées en langage formel OCL, et des descriptions textuelles informelles. La sémantique dynamique de VUML sera décrite sous forme de règles en langage naturel.

III.4.1. Sémantique statique de VUML

Dans cette section, nous décrivons la sémantique de chaque élément de modélisation introduit par VUML ainsi que les contraintes (règles de bonne modélisation) que doit vérifier cet élément. Afin de clarifier la description de chaque élément, nous allons présenter des extraits du méta-modèle associé au profil VUML en mettant l'accent à chaque fois sur l'élément concerné. Ensuite, les règles de bonne modélisation concernant l'élément (contraintes de l'utilisation du stéréotype correspondant à l'élément) sont décrites premièrement d'une manière textuelle et deuxièmement d'une manière formelle en OCL. De plus un exemple abstrait est présenté pour illustrer chaque règle.

D'autre part, nous étudions aussi dans cette section l'impact des concepts et mécanismes introduits par VUML sur la sémantique des différentes relations UML, à savoir, l'association, l'agrégation, la composition, et la spécialisation/généralisation. Ceci va nous permet de dégager un certain nombre de règles à respecter pour mener une modélisation VUML correcte.

Pour simplifier l'écriture formelle de certaines règles présentées dans cette section, nous avons défini un certain nombre de fonctions OCL génériques (cf. annexe B) :

- **allStereotypes** : elle retourne un ensemble contenant les stéréotypes de l'élément de modélisation courant et tous les stéréotypes des ancêtres de cet élément.

- **isStereotyped** : elle détermine si l'élément de modélisation courant est stéréotypé par le nom indiqué comme argument.
- **isStereokinded** : elle détermine si l'élément de modélisation courant est stéréotypé par le nom spécifié comme argument ou bien si l'un de ses ancêtres est stéréotypé par ce nom.
- **viewRoot** : elle détermine la racine de l'élément de modélisation spécifié comme argument. Cette racine est le premier ancêtre stéréotypé par « view » ou « abstractView » et source d'une dépendance « viewExtension ».
- **allParents** : elle retourne un ensemble contenant les ancêtres d'une classe.

III.4.1.1. Sémantique statique des éléments de modélisation introduits par VUML

Le profil VUML étend le méta-modèle UML en introduisant un certain nombre de nouveaux éléments de modélisation, à savoir : *Base*, *View*, *MultiViewsClass*, *ViewExtension*, *ViewDependency*, *MultiViewsComponent*, et *MVInterface*. Dans la suite de cette section nous décrivons la sémantique statique associée à chacun de ces éléments.

III.4.1.1.1. Base

C'est un élément de modélisation qui spécialise *GeneralView* qui est lui-même une spécialisation de l'élément de modélisation *Class*. Il décrit les caractéristiques structurelles et comportementales communes à tous les acteurs du *MultiViewsClass* dont il est composant. L'élément *Base* peut être associé à des relations de dépendance *ViewExtension* qui permettent de rattacher des vues à cette base (cf. figure 52).

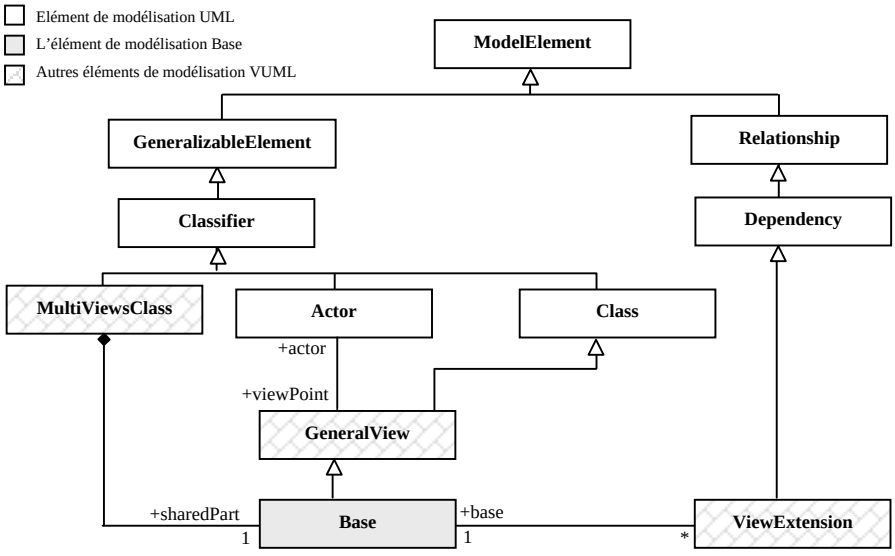

Figure 52 – *Extrait du métamodèle VUML : L'élément de modélisation Base*

Le stéréotype « base » est utilisé pour représenter la base d'une classe multivues. Donc dans un modèle VUML, on ne doit pas autoriser l'utilisation de ce stéréotype sur des classes qui n'ont pas de vues (c'est-à-dire les classes qui ne sont pas destination d'au moins une relation « viewExtension »). Cependant, tenant compte du mécanisme de l'héritage qui permet à une classe d'hériter les vues de sa classe parente, le stéréotype « base » est autorisé sur une classe qui hérite d'une « base » ou d'un « multiViewsClass ». Rappelons que ce dernier est utilisé pour représenter des classes multivues non

éclatées. Inversement, toute classe qui hérite d'une « base » doit être soit une « base » soit un « multiViewsClass ». D'où les règles de bonne modélisation suivantes :

Règles de bonne modélisation

[1] Une « base » a, au moins, une relation « viewExtension », ou doit être un descendant direct d'une « base » ou d'un « multiViewsClass ».

Formalisation en OCL

context base inv : self.supplierDependency->select(isStereotyped("viewExtension"))->size>=1 or self.isStereokinded("base") or self.isStereokinded("multiViewsClass")

L'exemple abstrait de la figure 53 montre que l'utilisation du stéréotype « base » sur la classe C est non valide, car cette classe n'est pas une descendante directe d'une « base » ou d'un « multiViewsClass » et n'a pas de relation « viewExtension ». Au contraire, les classes A, B et E stéréotypées par « base » sont valides. La classe A a trois relations « viewExtension », tandis que la classe B est une descendante directe d'un « multiViewsClass », et la classe E est une descendante directe d'une « base ».

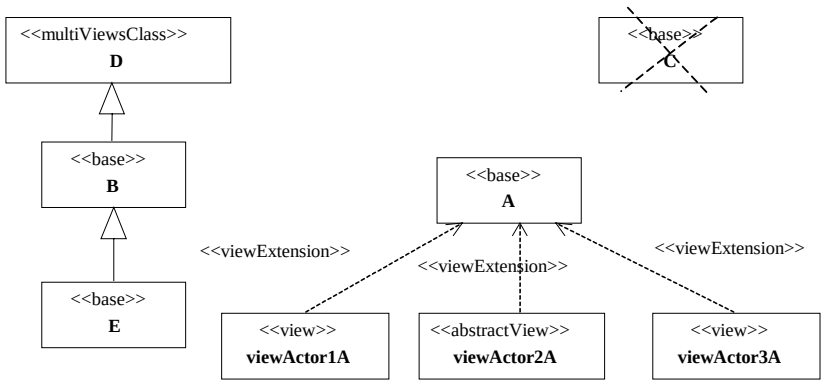

Figure 53 – *Illustration abstraite de l'utilisation de « base »*

[2] Un descendant direct de « base » est soit une « base » soit un « multiViewsClass ».

Formalisation en OCL

context base inv :
self.specialization->forAll (g : Generalization | g.child.isStereotyped("base") xor g.child.isStereotyped("multiViewsClass"))

Sur l'exemple abstrait de la figure 54, nous avons mis en évidence 3 spécialisations de la base A. La première, qui concerne la classe C, est non valide car C n'est stéréotypée ni par « base » ni par « multiViewsClass ». La deuxième spécialisation et la troisième spécialisation – concernant, respectivement, les classes E et F – sont valides car E est une « base » et F est un « multiViewsClass ».

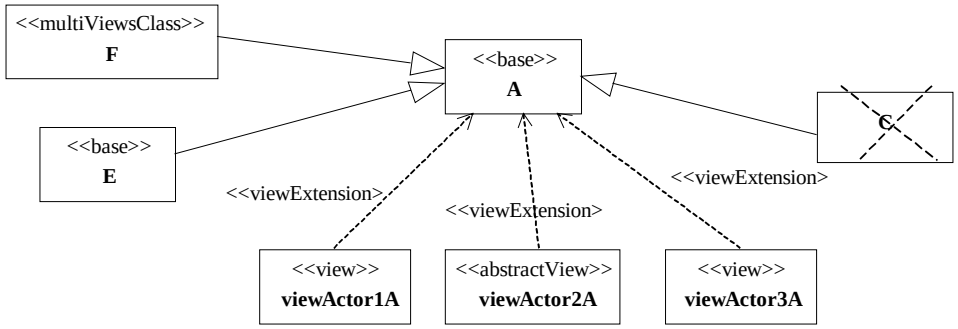

Figure 54 – *Illustration abstraite de la spécialisation de « base »*

III.4.1.1.2. View

C'est un élément de modélisation composant de l'élément de modélisation *MultiViewsClass*. Il spécialise *GeneralView*. L'élément *View* permet de modéliser les caractéristiques structurelles et comportementales spécifiques à un acteur donné tout en ayant la possibilité de redéfinir des caractéristiques de la *Base* à laquelle il est relié via la relation *ViewExtension*. Il peut aussi être source ou cible d'une au plusieurs relations *ViewDependency* (cf. figure 55).

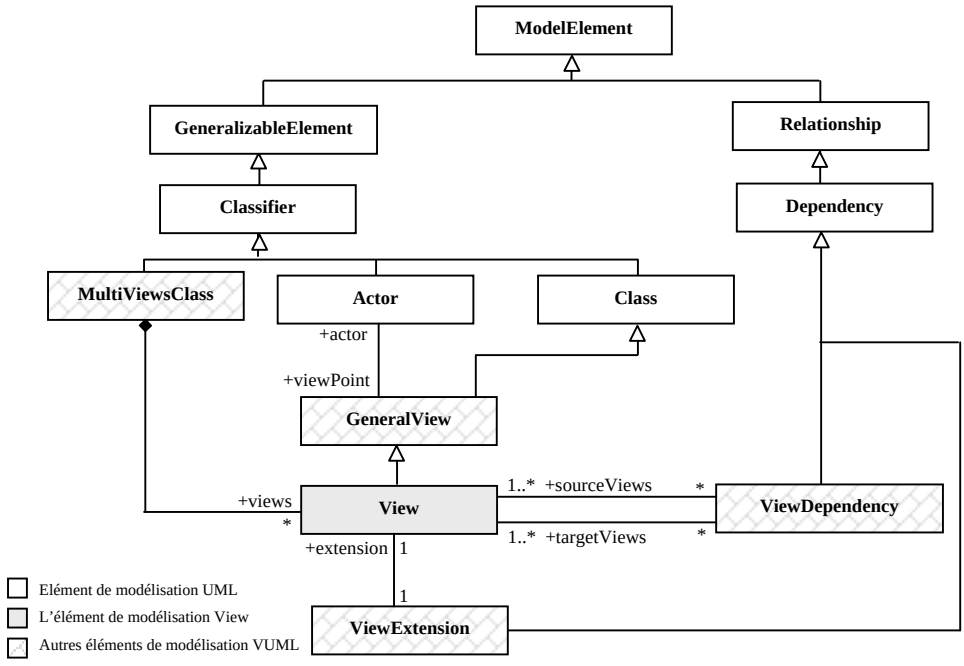

Figure 55 – *Extrait du métamodèle VUML : L'élément de modélisation View*

Dans VUML, le stéréotype « view » est utilisé pour représenter une vue. Nous avons vu dans le chapitre II qu'une vue ne peut pas avoir plus d'une classe parente. Cette contrainte qui vise à interdire l'héritage multiple est imposée dans le but de simplifier la structure d'une classe multivues. De plus la classe parente d'une vue ne peut être qu'une vue. Inversement, un descendant direct d'une vue est une vue. Cette vue peut être concrète ou abstraite. Le fait de spécialiser une vue pour la rendre abstraite peut être utilisé par exemple dans le cas où on veut créer des sous-vues de cette vue avec des méthodes

ayant les mêmes signatures mais avec des implémentations différentes. D'autre part, une vue d'une classe multivues peut spécialiser une vue d'une autre classe multivues. Cette spécialisation n'est possible que si la classe de la vue fille spécialise la classe de la vue parente (c'est-à-dire la base de la vue fille spécialise la base de la vue parente). Sémantiquement ceci n'est accepté que si l'acteur associé à la vue fille et aussi associé à la vue parente. Ces différentes contraintes font l'objet des règles de bonne modélisation suivantes :

Règles de bonne modélisation

[1] Un élément « view » a, au plus, un parent.

Formalisation en OCL

context view inv : self.generalization->size<=1

L'exemple de la figure 56 illustre le fait que les « view » *viewActor3A* et *viewActor11A* sont non valides car ils ont plus d'un parent. Le « view » *viewActor21A*, qui a un seul parent, est valide.

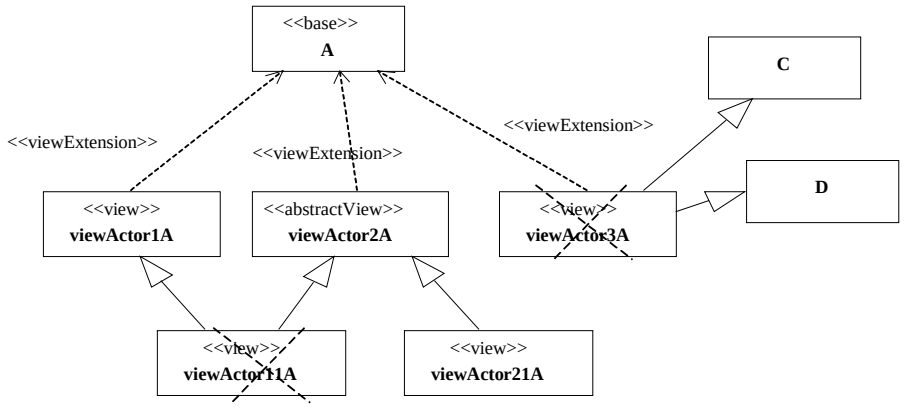

Figure 56 – *Illustration abstraite de l'utilisation de « view » avec héritage*

[2] Un élément « view » ne peut hériter que de « view » ou de « abstractView »

Formalisation en OCL

context view inv :
self.generalization->forAll (g : Generalization | g.parent.isStereotyped("view") or g.parent.isStereotyped("abstractView"))

Dans l'exemple de la figure 57, le « view » *viewActor3A* est non valide ; en effet cette classe hérite de la classe *C* qui n'est ni un « view » ni un « abstractView ». Les autres « view » ayant des parents, à savoir *viewActor1A, viewActor21A* et *viewActor31A* sont valides car leurs ancêtres sont soit des « view » soit des « abstractView ».

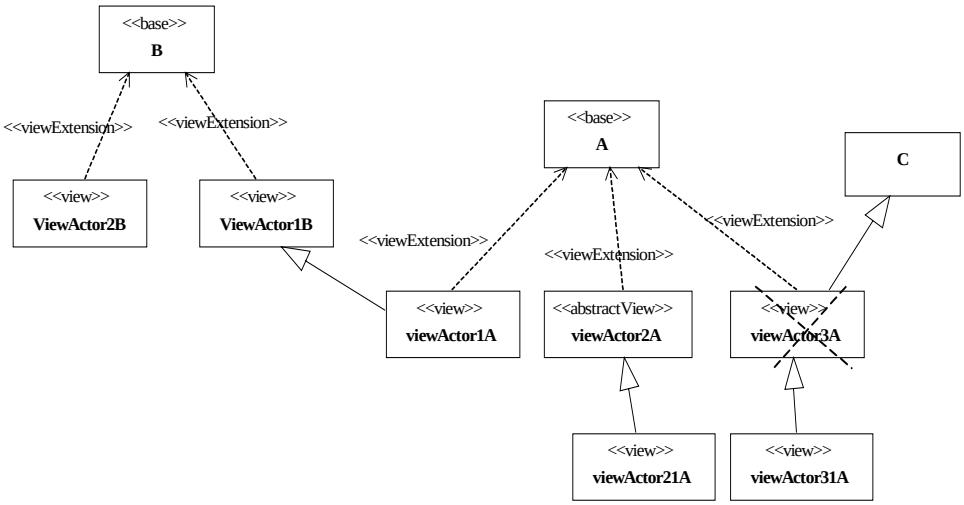

Figure 57 – *Illustration abstraite des ancêtres possibles de « view »*

[3] Un descendant direct d'un « view » est soit un « view » soit un « abstractView »

Formalisation en OCL

context view inv :
self.specialization->forAll (g : Generalization |g.child.isStereotyped("view") xor g.child.isStereotyped("abstractView"))

La figure 58 ci-après montre que la classe *F* est non valide car elle n'est stéréotypée ni par « view » ni par « abstractView » alors qu'elle est une spécialisation du « view » *viewActor1A*. Les autres spécialisations respectent bien la contrainte 3 ci-dessus.

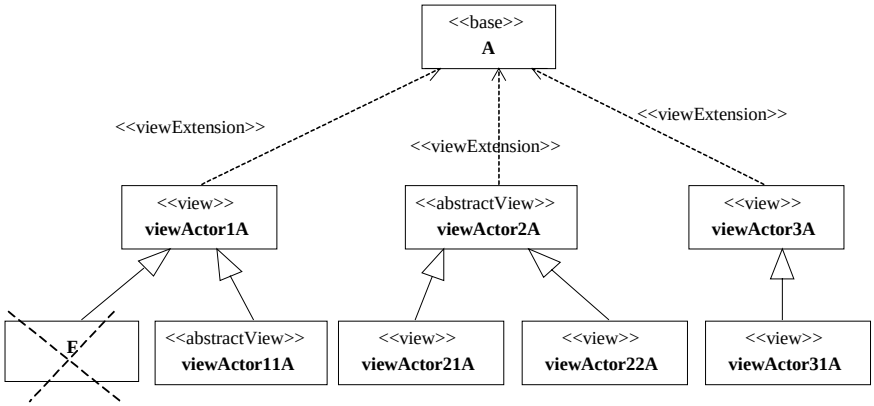

Figure 58 – *Illustration abstraite de la spécialisation de « view »*

[4] Un « view » doit être source d'une seule relation « viewExtension », ou être descendant d'un et un seul « view » ou d'un et un seul « abstractView ».

Formalisation en OCL

context view inv :
(self.clientDependency->select(isStereotyped("viewExtension"))->size=1) or
(self.isStereokinded("view") or self.isStereokinded("abstractView"))

Sur l'exemple de la figure 59, les « view » *viewActor2A* et *viewActor4A* sont non valides car le premier est source de deux relations « viewExtension », tandis que le deuxième n'est source ni d'une seule relation « viewExtension » ni descendant d'un « view » ou d'un « abstractView ». Les autres « view » présentés sur cet exemple se conforment bien à la règle 4 ci-dessus.

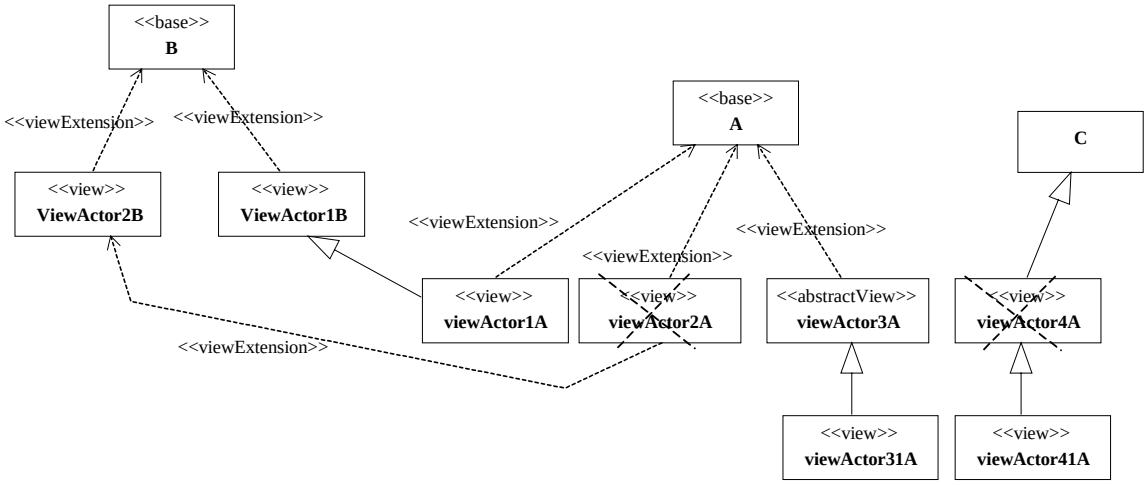

Figure 59 – *Illustration abstraite des contraintes que doit vérifier un « view »*

[5] Si un « view » est relié par une relation « viewExtension » à une base B1, et hérite d'un « view » ou « abstractView » d'une base B2, alors B1 est un descendant de B2.

Formalisation en OCL

context view inv :
self.clientDependency->select(isStereotyped("viewExtension"))->notEmpty and
-- récupération de la base de la vue dans B1
let B1=self.clientDependency-> select(isStereotyped("viewExtension")).supplier and

-- récupération de la racine du parent de la vue dans P
let P=self.viewRoot and
-- récupération de la base de la racine du parent de la vue dans B2
let B2=P.clientDependency-> select(isStereotyped("viewExtension")).supplier
implies
-- B1 est un descendant de B2
B1.allParents->exists(B2)

Sur la figure 60 ci-dessous, l'héritage entre *viewActor1B1* et *viewActor1B2* est valide car la base *B1* de *viewActor1B1* hérite de la base *B2* de *viewActor1B2*. Au contraire, l'héritage entre *viewActor4B1* et *viewActor1B3* est non valide car la base *B1* de *viewActor4B1* n'est pas un descendant de la base *B3* de *viewActor1B3*. De même l'héritage entre *viewActor2B1* et *viewActor3B1* qui ont la même base *B1* est non valide car la base *B1* n'est pas un descendant d'elle-même.

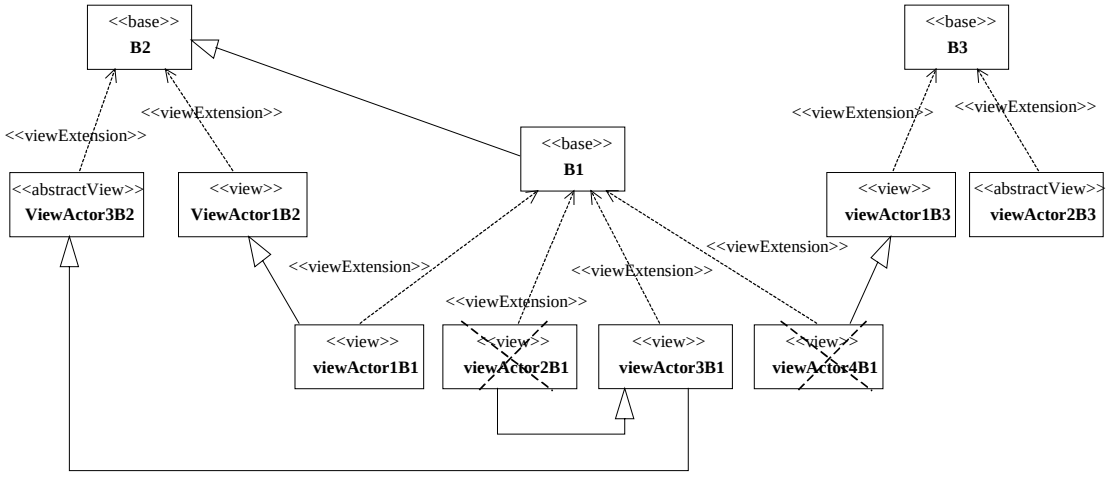

Figure 60 – *Illustration abstraite des contraintes induites par l'héritage entre des « view » sources de relations « viewExtension »*

Remarque : les règles de bonne modélisation concernant l'utilisation du stéréotype *« abstractView »* sont les mêmes règles que celles que doit vérifier l'utilisation du stéréotype *« view »*.

III.4.1.1.3. MultiViewsClass

L'élément de modélisation *MultiViewsClass* est une spécialisation de *Classifier*. Il est composé d'une *Base* et d'une liste de *View* reliés à la base via des *ViewExtension* (cf. figure 61). Le stéréotype correspondant « multiViewsClass » s'utilise pour représenter des classes multivues non éclatées sur le diagramme de classes. Conformément au mécanisme de l'héritage d'une classe multivues, les descendants d'un « multiViewsClass » sont par conséquent soit des « multiViewsClass » soit des « base ».

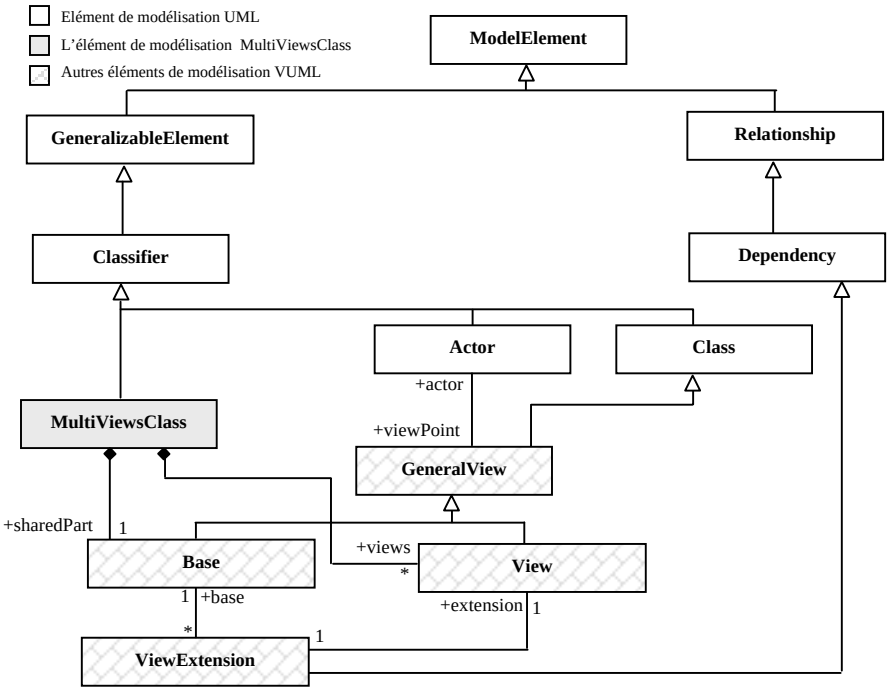

Figure 61 – *Extrait du métamodèle VUML : L'élément de modélisation « MultiViewsClass »*

Règle de bonne modélisation

[1] Un descendant direct de « multiViewsClass » est soit un « multiViewsClass » soit une « base ».

Formalisation en OCL

context multiViewsClass inv :
self.specialization->forAll (g : Generalization |
g.child.isStereotyped("multiViewsClass") xor g.child.isStereotyped("base"))

La classe *B3* présentée sur la figure 62 est non valide car elle n'est ni un « multiViewsClass » ni une « base » alors qu'elle hérite d'un « multiViewsClass ». Les classes *B1* et *B2*, qui sont des descendants du « multiViewsClass » *B*, sont valides car *B1* est une « base » et *B2* est un « multiViewsClass ».

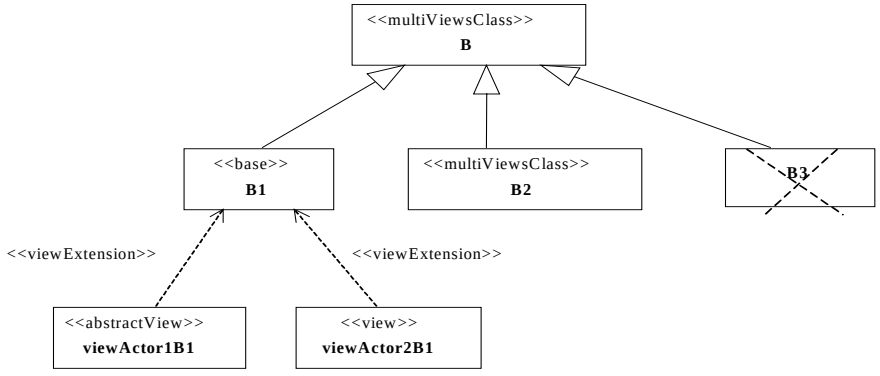

Figure 62 – *Illustration abstraite de la spécialisation de « multiViewsClass »*

III.4.1.1.4. ViewExtension

ViewExtension est un élément de modélisation spécialisant l'élément *Dependency* (cf. figure 63). C'est une dépendance ayant comme source une vue ou une vue abstraite et ayant comme cible une base. Elle permet de dupliquer, dans la source, la structure de la cible et les valeurs de ses attributs. *ViewExtension* n'est pas une relation d'héritage : les vues dépendent de la base au sens où les attributs et les méthodes de la base sont implicitement partagés par les vues de la classe multivues.

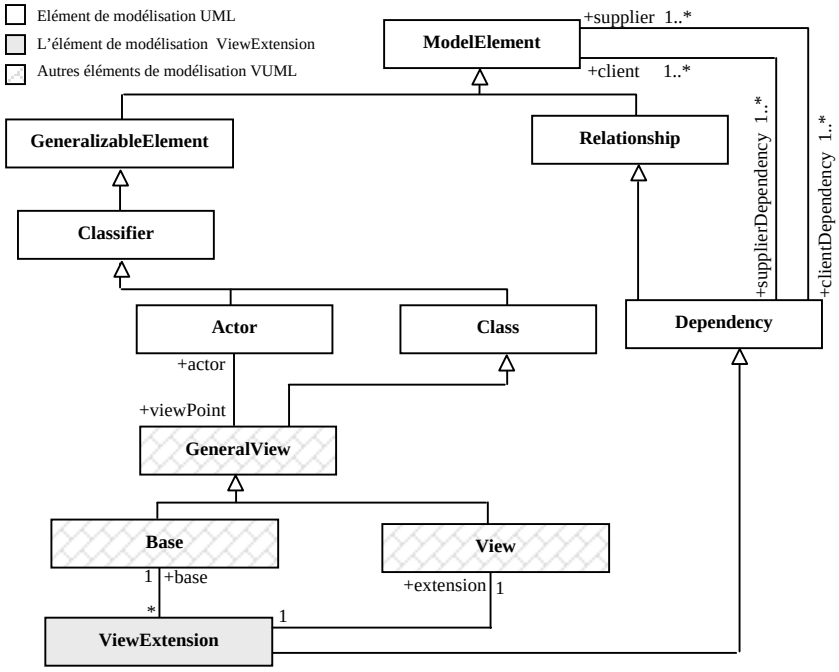

Figure 63 – *Extrait du métamodèle VUML : L'élément de modélisation « ViewExtension »*

Règle de bonne modélisation

[1] La dépendance « viewExtension » a pour source un « view » ou « abstractView » et a pour destination une « base ».

Formalisation en OCL

context viewExtension inv :
(self.client.isStereotyped("view") or self.client.isStereotyped("abstractView")) and
(self.supplier.isStereotyped ("base"))

L'exemple de la figure 64 montre deux relations « viewExtension » invalides. La première relation entre *viewActor21A* et *viewActor41A* est invalide car sa cible n'est pas une « base » ; la deuxième qui relie les classes *C* et *D* est aussi invalide car la source n'est ni un « view » ni un « abstractView » et la cible n'est pas une « base ». Les autres relations « viewExtension » respectent bien les conditions de la règle 1 ci-dessus.

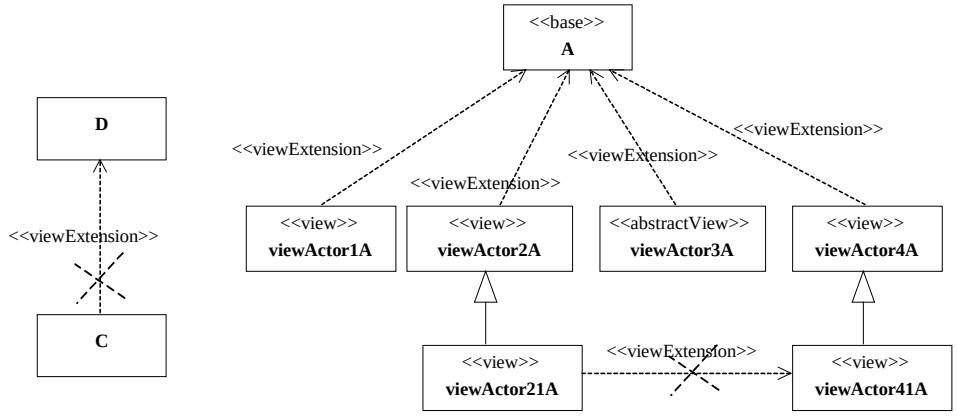

Figure 64 – *Illustration abstraite des contraintes que doit vérifier un « viewExtension »*

III.4.1.1.5. ViewDependency

ViewDependency est un élément de modélisation qui permet la modélisation des relations de dépendance entre les vues. Chaque *ViewDependency* peut être associé à une ou plusieurs contraintes (cf. figure 65). Ces contraintes expriment les dépendances fonctionnelles entre les vues reliées par la relation « viewDependency » concernée. Elles peuvent être exprimées soit en langage naturel, soit en langage formel tel que OCL.

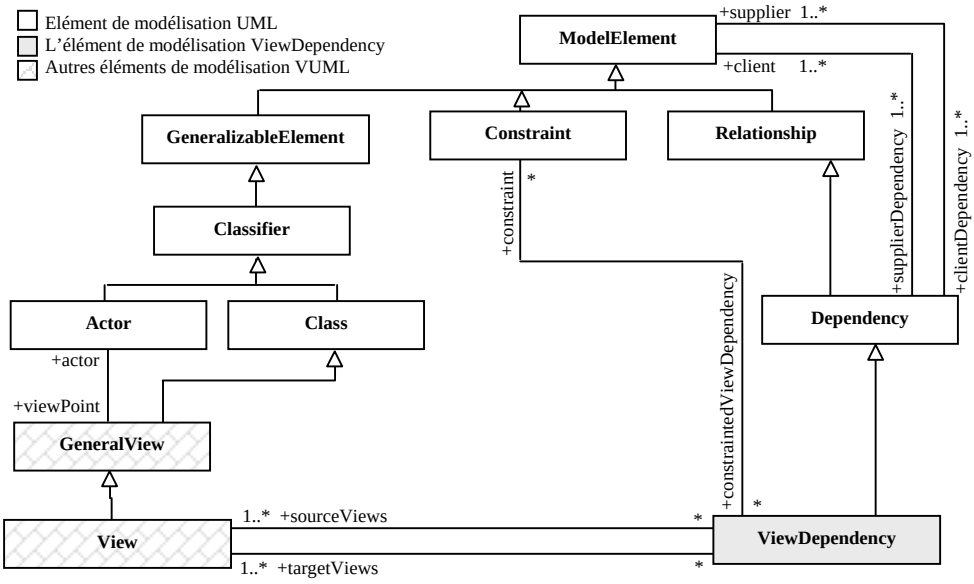

Figure 65 – *Extrait du métamodèle VUML : L'élément de modélisation « ViewDependency »*

La cible et la destination d'une dépendance « viewDependency » doivent être des vues (abstraites ou concrètes) différentes. Ces vues doivent appartenir soit à la même classe multivues soit à des classes multivues différentes. Dans ce dernier cas, la classe de la vue source doit être une descendante de la classe de la vue cible. En effet, ceci se justifie par le fait que le mécanisme de l'héritage entre des

classes multivues duplique virtuellement les vues de la classe parente dans la classe fille. Donc, virtuellement la dépendance relie toujours des vues appartenant à la même classe multivues.

Règles de bonne modélisation

[1] La dépendance « viewDependency » a pour source un « view » ou « abstractView » et a pour destination un « view » ou « abstractView ».

Formalisation en OCL

context viewDependency inv :
(self.client.isStereotyped("view") or self.client.isStereotyped("abstractView")) and
(self.supplier.isStereotyped("view") or self.supplier.isStereotyped("abstractView"))

L'exemple de la figure 66 illustre deux relations « viewDependency » invalides. La première relation qui relie les classes *C* et *D* est invalide car *C* et *D* ne sont ni des « view » ni des « abstractView ». Les autres relations « viewDependency » respectent la rège 1 ci-dessus.

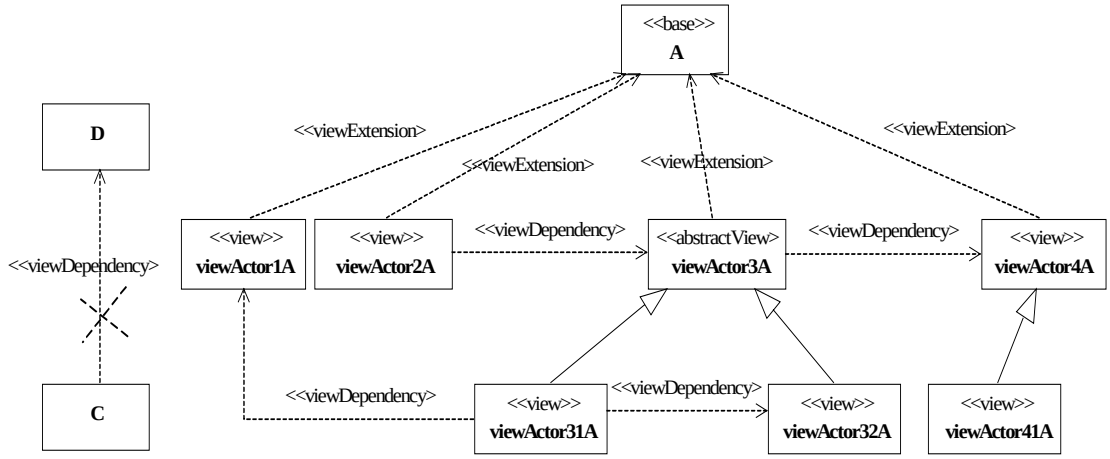

Figure 66 – *Illustration abstraite des contraintes que doit vérifier un « viewDependency »*

[2] Si la source d'une dépendance « viewDependency » a comme base B1 et la destination de cette même dépendance a comme base B2, alors soit B2=B1 soit B1 est un descendant de B2.

Formalisation en OCL

context viewDependency inv :
let B1=self.client.viewRoot.clientDependency->select(isStereotyped("viewExtension")).supplier and
let B2=self.supplier.viewRoot.clientDependency->select(isStereotyped("viewExtension")).supplier
implies
(B1=B2) xor (B1.allParents->exists(B2))

L'exemple abstrait présenté sur la figure 67 montre que la relation de dépendance entre la vue *viewActor1B1* et la vue *viewActor1B2* est valide car la base *B1* de *viewActor1B1* hérite de la base *B2*

de *viewActor1B2* ; tandis que la relation « viewDependency » ayant comme source *viewActor3B1* et comme cible *viewActor2B3* est non valide car la base *B1* n'hérite pas de la base *B3*.

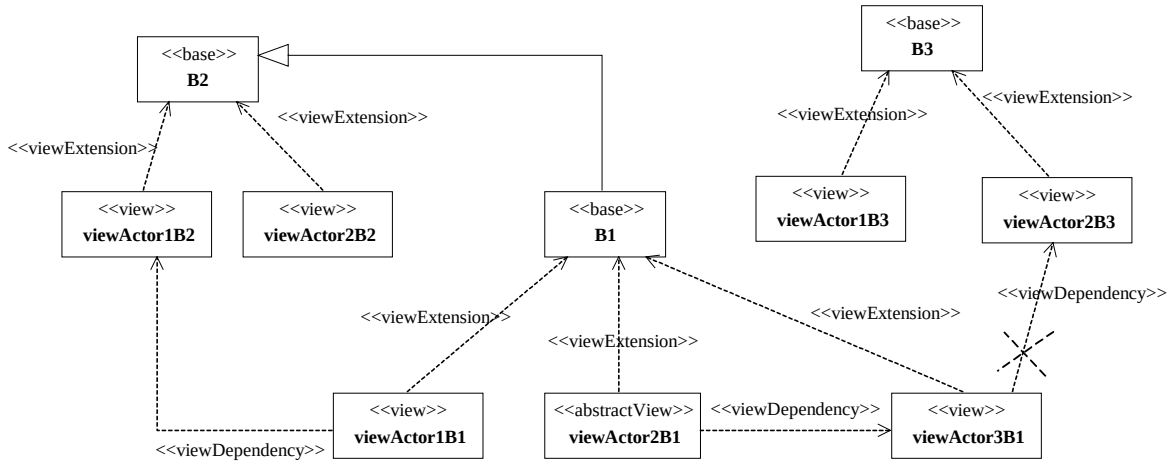

Figure 67 – *Illustration abstraite des contraintes induites par des relations « viewDependency » ayant la base de la source différente de la base de la cible.*

III.4.1.1.6. MultiViewsComponent

Nous avons vu dans le chapitre II (cf. section II.4) que VUML propose un nouveau modèle de composant appelé composant multivues. Ce nouveau modèle de composant est représenté en stéréotypant un composant UML par « multiViewsComponent ». Les règles 1 et 2 ci-après résument la sémantique statique que doit vérifier un « multiViewsComponent ».

Règles de bonne modélisation

[1] Un « multiViewsComponent » doit avoir une seule interface fournie I_SetView.

Formalisation en OCL

context MultiViewsComponent inv :
self.provided->select(name="I_SetView")->size=1

[2] Un « multiViewsComponent » doit avoir au moins une interface multivues.

Formalisation en OCL

context MultiViewsComponent inv :
self.provided->select(isStereotyped("MVInterface"))->size +
self.required->select(isStereotyped("MVInterface"))->size>=1

III.4.1.1.7. MVInterface

Une interface multivues (*MVInterface)* est une spécialisation d'une interface classique. Sa particularité réside dans le fait qu'elle n'est accessible qu'après l'activation de l'une des vues qui sont associées à cette interafce (cf. section II.4.1, chapitre II). Structurellement, une interface multivues doit vérifier les règles suivantes :

Règles de bonne modélisation

[1] Un « MVInterface » ne peut être qu'une interface d'un « multiViewsComponent »

Formalisation en OCL

context MVInterface inv :
self.owner.isStereotyped("multiViewsComponent")

[2] Un « MVInterface » est associé à au moins un acteur.

Formalisation en OCL

context MVInterface inv :
self.actors->size >=1

III.4.1.2. Sémantique statique des relations UML usuelles dans VUML

Dans cette section, nous étudions l'impact du concept de classe multivues sur la sémantique des différentes relations UML. Nous dégageons par la suite des règles de bonne modélisation concernant l'utilisation de ces relations.

Rappelons qu'une vue est associée à un acteur et correspond à l'application du point de vue de cet acteur sur une entité, et que la base d'une classe multivues représente la partie partagée entre les différentes vues de cette classe multivues. Donc, seuls les acteurs ayant des vues sur une classe multivues peuvent accéder à la base de cette classe multivues. Rappelons aussi qu'une classe publique est une classe qui est accessible par tous les acteurs du système. Les règles de bonne modélisation que nous présentons dans cette section tiennent compte de ces principes. Pour chaque type de relation (association, agrégation, composition, spécialisation/généralisation), nous étudions trois situations : relation base-base, relation base-vue, relation vue-vue, relation classe publique-base, relation classe publique-vue.

III.4.1.2.1. Relation d'association

L'association en UML exprime une relation à couplage faible, les classes associées restent relativement indépendantes l'une de l'autre (relation symétrique). Dans VUML, la sémantique de cette relation n'est pas modifiée. Cependant, un certain nombre de restrictions est imposé sur l'utilisation de la relation d'association afin de prendre en compte les droits d'accès aux informations et services encapsulés dans les vues.

Règles de bonne modélisation

La relation d'association entre deux bases B1 et B2 n'est autorisée que s'il existe un acteur associé à une vue de la classe B1 et associé aussi à une vue de la classe B2 ce qui justifie la possibilité d'avoir des accès entre les deux bases. De même une relation d'association reliant une base et une vue n'est permise que si l'acteur associé à la vue est aussi associé à une vue de cette base. En effet, dans le cas contraire, cette relation d'association n'aurait pas de sens dans la mesure où aucun accès ne serait possible entre la vue et la base. Ces deux restrictions sont formalisées dans les règles 1 et 2.

[1] Soit R une association entre deux bases B1 et B2. R est valide si et seulement s'il existe un acteur A associé à une vue de B1 et associé à une vue de B2.

Formalisation en OCL

```
Context Association inv :
Let A=self.connection->select(participant.isStereotyped("base")) and
A->size>1  and
-- récupération des deux bases participant à l'association
Let base1=A.first and
Let base2=A.last and
-- récupération de la liste des vues de base1
Let views1=base1.supplierDependency->select(isStereotyped("viewExtension")).client and
-- récupération de la liste des vues de base2
Let views2=base2.supplierDependency->select(isStereotyped("viewExtension")).client and
-- récupération de l'intersection entre views1 et views2 : un élément V1  de views1 appartient à cette
--  intersection si et seulement si l'acteur associé à V1 a une vue dans views2
Let intersection =views1->select(v1|views2->select(v2|v2.actor=v1.actor)->notEmpty)
implies
intersection->notEmpty
```

[2] Soit R une association entre une base B et une vue V d'une autre base et soit A l'acteur associé à la vue V. R est valide si et seulement si l'acteur A est associé à une vue de B.

Formalisation en OCL

```
Context Association inv :
Let C1=self.connection->select(participant.isStereotyped("base")) and
Let C2=self.connection->select(participant.isStereotyped("view") or
participant.isStereotyped("abstractView"))  and
C1->size=1  and
C2->size=1 and
-- récupération de la base participante à l'association
Let base=C1.first  and
 -- récupération de l'acteur associé à la vue participant à l'association
Let acteur=C2.first.actor  and
-- récupération de la liste des vues de la base participant à l'association
Let views=base.supplierDependency->select(isStereotyped("viewExtension")).client
Implies    views->select(v|v.actor=acteur)->notEmpty
```

En ce qui concerne l'association ente les vues (abstraites ou concrètes), ce type d'association est interdit car chaque vue correspond aux besoins et droits d'accès spécifiques d'un acteur donné. Une telle association violerait cette caractéristique. De plus toute utilisation de vues doit passer par la base de la classe multivues de ces vues (aucun accès direct aux vues n'est autorisé). Donc, la relation d'association entre deux vues est interdite même si ces deux vues sont associées au même acteur.

Règles de bonne modélisation

[3] Toute association entre des vues concrètes ou abstraites est interdite.

Formalisation en OCL

Context Association inv :
self.connection->select(participant.isStereotyped("view") or
participant.isStereotyped("abstractView"))->size<=1

Concernant la relation d'association entre les vues et les classes publiques, cette relation est autorisée mais elle doit être uniquement navigable dans le sens vues-classes. En effet, ceci signifie que les vues auront accès aux objets de ces classes mais pas l'inverse, ce qui est conforme à la définition d'une vue qui n'est pas accessible directement.

Les relations d'association reliant des vues ou des bases à des classes publiques sont autorisées. En effet, une classe publique peut être considérée comme une base ayant des vues (vides) associées à tous les acteurs du système. Donc, dans ce cas les règles 1 et 2 sont vérifiées. Cependant et afin d'interdire l'accès direct à une vue, toute association qui fait intervenir une vue doit être navigable uniquement à partir de cette vue.

[4] Si une vue est reliée par une association avec une classe C (qui n'est pas une vue) alors cette association doit être uniquement navigable dans le sens vue vers C.

Formalisation en OCL

Context Association inv :
Let A=self.connection->select(participant.isStereotyped("view") or
participant.isStereotyped("abstractView"))
A->size=1 implies
self.connextion->select(ae|ae<>A->first)forAll(ae|ae.isNavigable=#true)

Sur la figure 68, l'association B2-B4 est non valide car il n'y a aucun acteur qui ait à la fois une vue sur B2 et une vue sur B4 (règle 1). De même l'assocation viewActor3B3-B5 est non valide car l'acteur « Actor 3 » associé à la vue viewActor3B3 n'a pas une vue sur B5. Les relations d'association viewActor1B1-viewActor1B2, viewActor3B3-viewActor4B3 et viewActor1B1-C3 sont aussi non valides. En effet, les deux relations d'association viewActor1B1-viewActor1B2 et viewActor3B3-viewActor4B3 sont des relations vue-vue (règle 3), tandis que la relation viewActor1B1-C3 est une relation vue-classe mais navigable dans les deux directions (règle 4). Les autres relations d'association, à savoir B1-B2, B1-C1, viewActor1B1-C2, viewActor3B1-B3 respectent bien les règles ci-dessus.

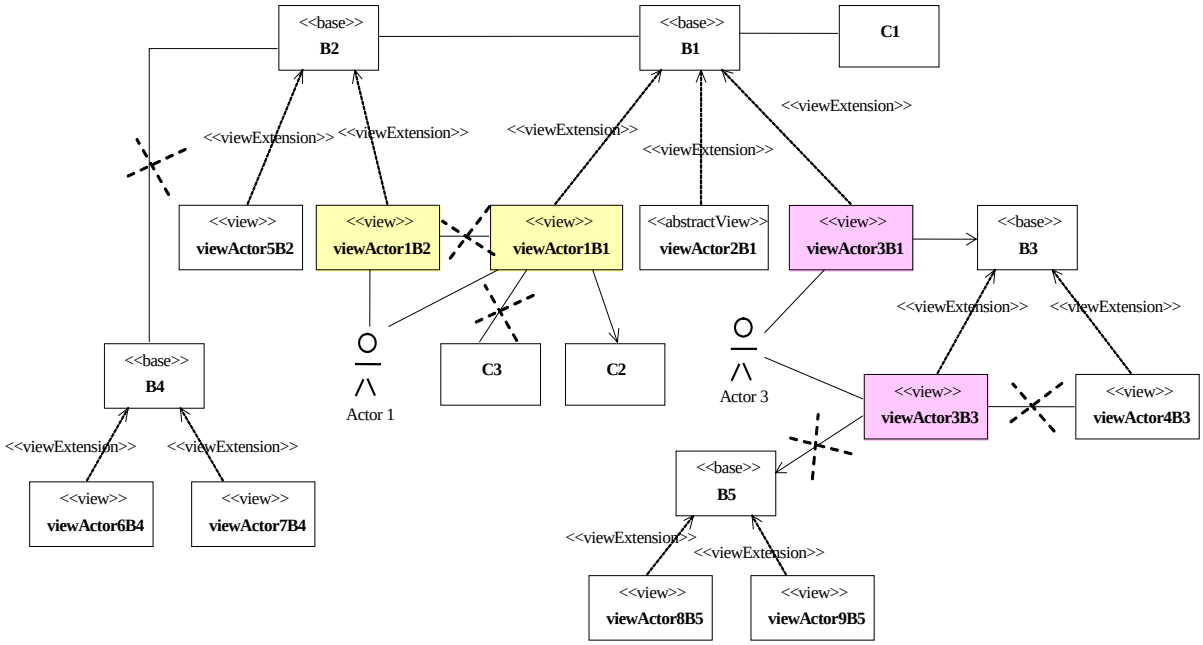

Figure 68 – *Illustration abstraite de la relation d'association dans VUML*

III.4.1.2.2. Relation d'agrégation

Une agrégation est une forme particulière d'association qui exprime un couplage plus fort entre classes. Elle matérialise une relation de type maître/esclave, transitive, non symétrique et réflexive. L'agrégation favorise la propagation des valeurs d'attributs et des opérations vers les agrégés.

Comme la relation d'association, l'approche VUML n'a pas modifié la sémantique de la relation d'agrégation, mais son utilisation est soumise à un certain nombre de restrictions. En plus des règles de bonne modélisation concernant la relation d'association, l'utilisation de l'agrégation doit respecter les règles 1 et 2 ci-dessous. La première règle reflète toujours le fait qu'une vue ne doive pas être accessible directement sans passer par sa base. Par conséquent, une vue ne doit pas être l'agrégé (esclave) dans une relation d'agrégation. La deuxième règle rejoint la règle 4 concernant la relation d'association mais ici la vue joue le rôle d'agrégat. Dans ce cas la relation d'agrégation doit être navigable à partir de la vue vers la classe publique ce qui garantit qu'il n'aura aucun accès à la vue dans la classe agrégée.

Règles de bonne modélisation

[1] Une vue ne peut jamais jouer le rôle d'agrégé dans des relations d'agrégation.

Formalisation en OCL

Context Association inv :
Let A=self.connection->select(aggregation=#aggregate)
A->size=1 implies
self.connection->select(ae|ae<>A->first)->forAll(ae| not ae.participant.isStereotyped("view") and not ae.participant.isStereotyped("abstractView"))

[2] Si une vue est reliée par une relation d'agrégation à une classe C (qui n'est pas une vue), alors la classe C doit être l'agrégé de cette relation, et cette agrégation doit être uniquement navigable dans le sens vue vers C.

Formalisation en OCL

Context Association inv :
Let A=self.connection->select((participant.isStereotyped("view") or
participant.isStereotyped("abstractView")) and aggregation=#aggregate)
A->size=1 implies
self.connextion->select(ae|ae<>A->first)->forAll(ae|ae.isNavigable=#true)

Sur la figure 69 ci-après, les relations d'agrégation viewActor2B1-viewActor3B1, viewActor1B1-viewActor1B2, et viewActor5B2-B4 sont non valables (règle 1). La relation d'agrégation viewActor1B1-C3 est aussi non valable (règle 2). Par contre l'agrégation B1-B2 est valide (règel 1 concernant l'association). De même l'agrégation viewActor3B1-B3 est valide (règle 2 propre à l'agrégation et règle 2 concernant l'association).

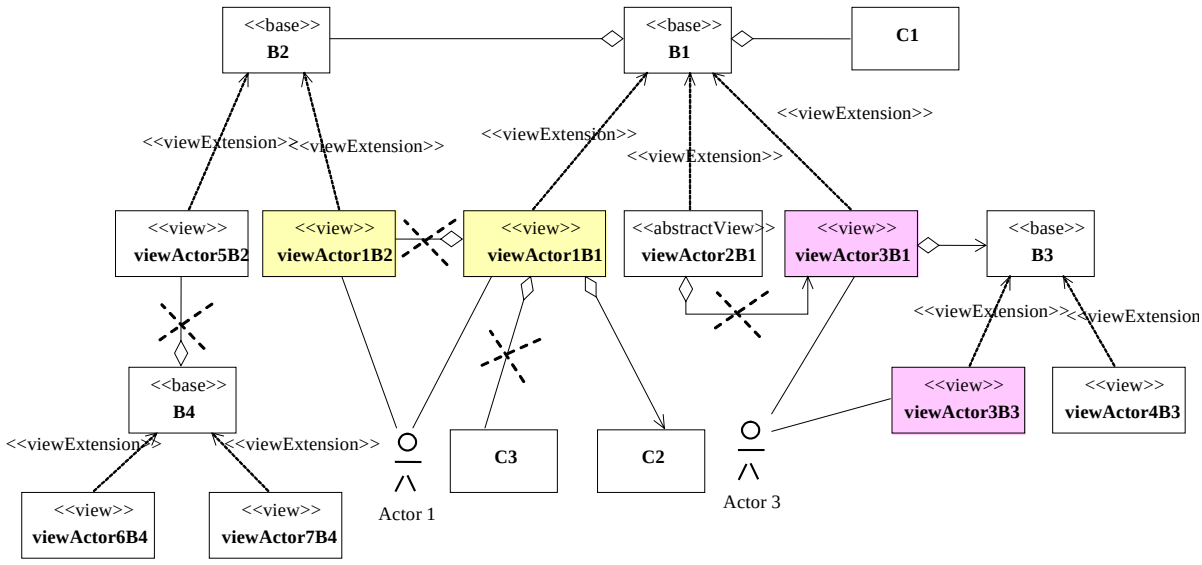

Figure 69 – *Illustration abstraite de la relation d'agrégation dans VUML*

III.4.1.2.3. Relation de composition

La sémantique de la relation de composition d'UML n'est pas modifiée par VUML. Cette relation constitue toujours une forme d'agrégation avec un couplage plus important. La composition implique une coïncidence des durées de vie des composants et du composite. La création, la modification et la destruction des composants sont de la responsabilité du composite. Cependant, en plus des règles concernant l'agrégation qui restent valables pour la relation de composition, VUML impose d'autres règles à respecter pour une bonne utilisation de la relation de composition dans les modèles VUML. Ces règles tiennent compte des particularités de cette relation de composition. La règle 1 ci-dessous concerne la prise en compte des droits d'accès aux vues, tandis que les règles 2 et 3 prennent en

considération le fait que les acteurs qui ont des vues sur un composant doivent aussi avoir des vues sur son composite. De plus une vue composite ne peut pas avoir de composants ayant une vue associée à un acteur autre que l'acteur associé à la vue composite.

Règles de bonne modélisation

[1] Une vue ne peut jamais être un composant dans une relation de composition.

Formalisation en OCL

Context Association inv :
Let A=self.connection->select(aggregation=#composite)
A->size=1 implies
self.connextion->select(ae|ae<>A->first)->forAll(ae| not ae.participant.isStereotyped("view") and not ae.participant.isStereotyped("abstractView"))

[2] Soit R une relation de composition reliant une base B1 (composite) à une base B2 (composant), et soient V1, V2, …, Vn les vues de la base B2 associées, respectivement, aux acteurs A1, A2, …, An. La relation R est valide si et seulement si les acteurs A1, A2, …, An ont des vues sur la base B1.

Formalisation en OCL

Context Association inv :
Let C1=self.connection-> select(aggregation=#composite) and
Let C2=self.connection-> select(aggregation<>#composite) and
C1->size=1 and
C2->size=1 and
-- récupération des deux bases participant à l'association
Let base1=C1.first and
Let base2=C2.first and
-- récupération de la liste des vues de base1
Let views1=base1.supplierDependency->select(isStereotyped("viewExtension")).client and
-- récupération de la liste des vues de base2
Let views2=base2.supplierDependency->select(isStereotyped("viewExtension")).client
implies
-- L'acteur de chaque élément de views2 est un acteur d'un élément de views1
views2->forAll(v2|views1->select(v1|v1.actor=v2.actor)->notEmpty)

[3] Soit R une relation de composition reliant une vue V1 (composite), associée à un acteur A1, à une base B1 (composant). Et soit S l'ensemble des vues de B2. La relation R est valide si et seulement si S est réduit à une seule vue associée à l'acteur A1.

Formalisation en OCL

Context Association inv :
Let C1=self.connection-> select(aggregation=#composite) and
Let C2=self.connection-> select(aggregation<>#composite) and
C1->size=1 and

C2->size=1 and
-- récupération de la vue participant à la composition
Let view=C1.first and
-- récupération de la base participant à la composition
Let base1=C2.first and
-- récupération de la liste des vues de base1
Let views1=base1.supplierDependency->select(isStereotyped("viewExtension")).client
implies
-- views1 est réduit à un seul élément dont l'acteur est celui de la vue view
views1->size=1 and
views1.first.actor=view.actor

Sur la figure 70 les relations de composition viewActor2B1-viewActor3B1, viewActor1B1-viewActor1B2, et B4-viewActor5B2 ne sont pas valides (règle 1). La relation de composition B1-B2 est aussi invalide car l'acteur (Actor 5) n'a pas de vue sur B1 (règle 2). De même la relation de composition viewActor3B1-B3 est non valide (règle 3).

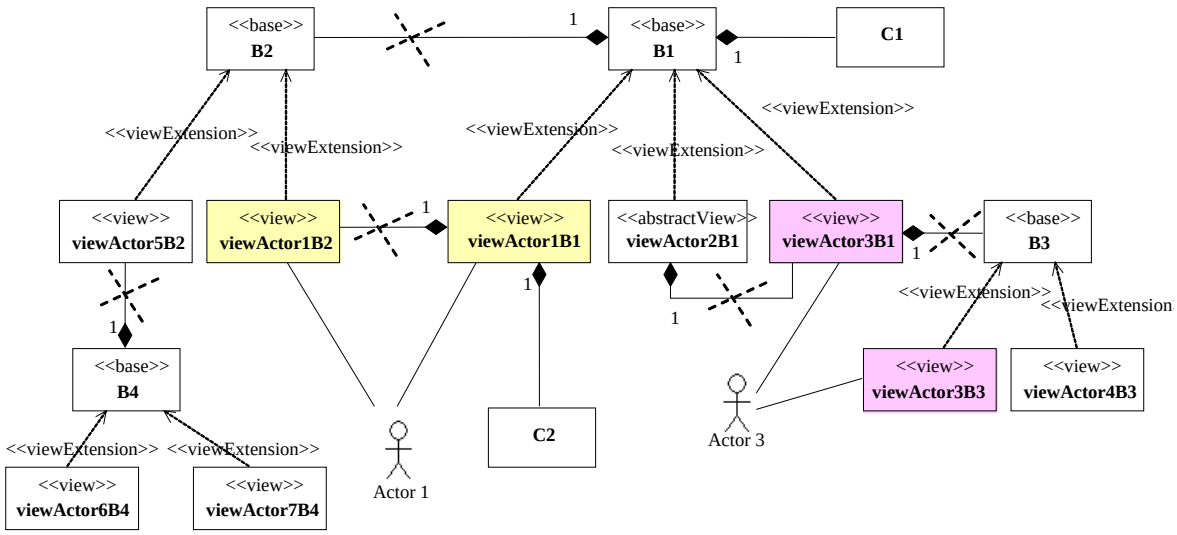

Figure 70 – *Illustration abstraite de la relation de composition dans VUML*

III.4.1.2.4. Relation de généralisation/spécialisation

La généralisation n'est pas modifiée par VUML. Elle constitue toujours une relation taxonomique entre un élément plus général et un élément plus spécifique. Comme nous l'avons dit dans le chapitre II (section II.3.4), une classe multivues peut être spécialisée par une autre classe. Dans ce cas, la classe fille devient automatiquement multivues. Les vues de la classe mère deviennent des vues de la classe fille. Cette dernière a la possibilité de redéfinir les vues de la classe mère et d'ajouter d'autres vues. La classe la plus générale et la classe la plus spécifique peuvent être des classes abstraites. Dans ce cas, leurs bases doivent être abstraites.

VUML impose un certain nombre de contraintes à respecter pour mener une modélisation cohérente en utilisant la relation de généralisation/spécialisation. La plupart de ces contraintes ont été présentées et formalisées en OCL dans la section III.4.1.1 et concernent les éléments : *Base*, *View*,

MultiViewsClass, *ViewDependency* (contrainte 2). De plus, afin de matérialiser la redéfinition d'une vue d'une classe multivues mère dans une de ses classes filles, la contrainte structurelle suivante est imposée (contrainte 1). La contrainte 2 n'autorise la redéfinition d'une vue que si la vue fille est associée à un acteur ayant un point de vue incluant celui de l'acteur de la vue à redéfinir.

Règles de bonne modélisation

[1] Soit B1 une base ayant une vue V1 associée à l'acteur A1 et soit B2 une base spécialisant la base B1. Si la base B2 a une vue V2 associée à l'acteur A1, alors la vue V2 doit être une spécialisation de la vue V1.

Formalisation en OCL

Context View inv :
-- récupération de la base de la vue courante dans B1
let B1=self.viewRoot.clientDependency-> select(isStereotyped("viewExtension")).supplier and
-- récupération de la liste des bases descendantes de B1
Let BasesFilles=(self.specialization->select(child.isStereotyped("base")))->collect(child) and
BasesFilles.forAll(B|B.supplierDependency->select(isStereotyped("viewExtension")).client->
forAll(v|v.actor=self.actor implies v.generalization.parent->exists(self)))

[2] Soit R une relation d'héritage entre deux vues V1 et V2 (V2 hérite de V1), et soient B1 et B2 (B2 hérite de B1) les bases, respectivement, de V1 et V2. La relation R est valide si et seulement si le point de vue de l'acteur associé à la vue V2 inclut le point de vue de l'acteur associé à la vue V1.

Formalisation en OCL

Context Generalization inv :
-- récupération des participants à cette généralisation
Let V1 = self.parent and
Let V2 = self.child and
-- V1 et V2 doivent être des vues (concrètes ou abstraites)
(V1.isStereotyped("view") or V1.isStereotyped("abstractView")) and (V2.isStereotyped("view") or V2.isStereotyped("abstractView")) and
-- récupération des deux bases de V1 et V2
let B1=V1.viewRoot.clientDependency-> select(isStereotyped("viewExtension")).supplier and
let B2=V2.clientDependency-> select(isStereotyped("viewExtension")).supplier and
-- B2 doit être un descendant de B1
B2.generalization->collect(parent)->exists(B1)
implies
V2.actor.generalization->collect(parent)->exists(V1.actor)

L'exemple de la figure 71 met en évidence des relations d'héritage entre vues. La première entre viewActor1B2 et viewActor1B1 est valide car les deux vues concernent le même acteur (Actor1). La deuxième entre la vue viewActor4B2 et la vue viewActor2B1 est non valide car elle ne vérifie pas la contrainte 2 : la première vue concerne l'acteur (Actor4) et la deuxième concerne l'acteur (Actor2).

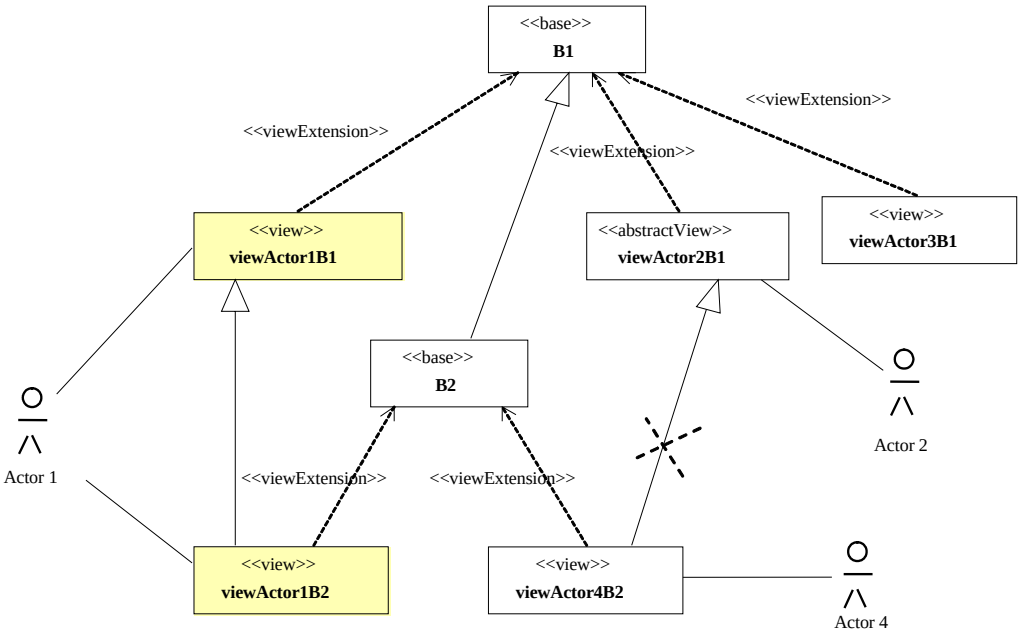

Figure 71 – *Illustration abstraite de la relation de généralisation/spécialisation dans VUML*

III.4.2. Sémantique dynamique de VUML

Après la présentation des différentes règles concernant la sémantique statique de VUML, nous abordons dans cette section les règles relatives à sa sémantique dynamique. Cette sémantique dynamique concerne particulièrement les mécanismes introduits par VUML, à savoir : activation/désactivation des vues, blocage/déblocage des vues, répercussion des modifications, et propagation de la vue active. Elle concerne également l'instanciation, le traitement des messages, et le polymorphisme qui sont des mécanismes UML mais leur sémantique est légèrement modifiée afin de supporter la notion de classe multivues introduite par VUML.

La sémantique dynamique de VUML est présentée sous forme de règles en langage naturel (Français) mais nous pensons que l'utilisation d'un langage formel serait intéressante et éviterait certaines mauvaises interprétations.

III.4.2.1. Instanciation

Dans VUML, l'instanciation garde la même sémantique que celle d'UML. Cependant, VUML impose des contraintes à respecter lors de l'instanciation d'une classe multivues. Une telle instanciation doit toujours débuter par l'instanciation de la base de la classe multivues (qui doit être concrète). Cette instanciation de la base permet de créer la partie partagée de l'objet accessible par tous les acteurs de la classe multivues. Elle permet aussi de créer une vue d'administration permettant la gestion des vues de cette classe multivues. Cette vue d'administration offre la possibilité d'effectuer une initialisation des vues qui consiste à préciser les arguments effectifs exigés par chacun des constructeurs des vues. Toute utilisation de vue n'est permise qu'après la réalisation de cette initialisation. Ces contraintes sont résumées dans les règles 1, 2, 3 et 4 ci-dessous.

[1] Aucune instanciation de vues n'est possible avant l'instanciation de la base de la classe multivues.

[2] Aucune instanciation directe de vues d'une classe multivues n'est permise.

[3] L'initialisation des vues doit se faire à travers la vue d'administration.

[4] L'instanciation d'une base abstraite ou d'une vue abstraite est non permise.

III.4.2.2. Activation/désactivation de vues

L'activation/désactivation des vues sont deux mécanismes pris en charge par l'accesseur *setView()*. Ce dernier permet de désactiver la vue courante avant d'activer la vue demandée. L'activation d'une vue doit respecter quelques contraintes résumées dans les deux règles suivantes :

[1] L'activation d'une vue nécessite la satisfaction des conditions suivantes :
- L'existence de la vue
- L'initialisation de la vue
- La vue n'est pas une vue bloquée.

[2] La désactivation d'une vue exige que la vue soit déjà active.

III.4.2.3. Répercussion des modifications

La répercussion des modifications est un mécanisme qui permet la mise en cohérence des différentes vues d'un objet multivues. Cette mise en cohérence se fait avant d'activer une vue. Elle consiste à répercuter les modifications de la vue active vers les vues dépendantes de cette vue active. Ces vues doivent être déjà créées (pour des raisons d'optimisation, une vue n'est créée que lors de sa première activation).
[1] Soient V1 et V2 deux vues dépendantes. La répercussion des modifications de la vue V1 vers la vue V2 doit vérifier les contraintes suivantes :
- V1 doit être active,
- V2 doit être créée.

III.4.2.4. Blocage/Déblocage des vues

Le blocage/déblocage des vues sont des opérations accessibles à travers la vue d'administration. Elles permettent de bloquer des vues, et de débloquer les vues qui ont été bloquées. La bonne exécution de ces opérations dépend du respect des règles suivantes :

[1] Le blocage d'une vue V exige que :
- V existe,
- V n'est pas une vue bloquée.

[2] Le déblocage d'une vue V exige que :
- V existe,
- V est une vue bloquée.

III.4.2.5. Propagation de la vue active

Sémantiquement, le fait d'activer une vue d'un objet multivues peut être considéré comme un éclairage selon cette vue sur tout l'objet y compris sur les objets multivues reliés à cet objet via des liens d'association, d'agrégation ou de composition. Ceci est pris en charge dans VUML par le mécanisme de propagation de la vue active (cf. chapitre II, section II.5.3). Ce mécanisme consiste à activer la vue active d'un objet multivues sur ses objets associés, ses objets agrégés, et ses objets composants. Dans la suite nous donnons les règles auxquelles obéit ce mécanisme de propagation de la vue active.

Lien d'association et lien d'agrégation

[1] Soit L un lien d'association ou d'agrégation reliant deux bases B1 (agrégat en cas d'agrégation) et B2 (agrégé en cas d'agrégation), et soient V1 une vue de B1 et V2 une vue de B2. Si V1 et V2 sont des vues associées au même acteur, alors, l'activation/désactivation de la vue V1 entraîne l'activation/désactivation de la vue V2.

[2] Soit L un lien d'association ou d'agrégation reliant une vue V1 (agrégat en cas d'agrégation) – associée à un acteur A – et une base B (agrégé en cas d'agrégation). Soit V2 une vue de B. Si la vue V2 est associée à l'acteur A, alors, l'activation/désactivation de la vue V1 entraîne l'activation/désactivation de la vue V2.

Lien de composition

[1] Soit L un lien de composition reliant deux bases B1 (composite) et B2 (composant), et soient V1 une vue de B1 et V2 une vue de B2. Si V1 et V2 sont des vues associées au même acteur, alors, toute opération (activation, désactivation, blocage, déblocage) faite sur la vue V1 sera aussi faite sur la vue V2.

[2] Soit L un lien de composition reliant une vue V1 (composite) – associée à un acteur A – et une base B (composant). Soit V2 la vue de B, alors toute opération (activation, désactivation, blocage, déblocage) faite sur la vue V1 sera aussi faite sur la vue V2.

III.4.2.6. Traitement des messages

Le mécanisme de traitement des messages n'est pas fondamentalement modifié par VUML. Un objet multivues traite les messages d'une manière analogue à celle utilisée par un objet classique. La seule différence est que l'objet multivues change son comportement selon la vue active ; par conséquent la réponse à un message à un instant donné dépend de la vue active à cet instant. Le traitement d'un message par un objet multivues se fait en respectant les règles suivantes :

[1] Soit *f* une méthode d'une vue V1. L'appel de la méthode *f* nécessite que la vue V1 soit active.

[2] Soit *f* une opération supportée par un objet multivues *O*. Un appel de *f* se fait de la manière suivante :

- recherche de *f* dans la vue active de l'objet *O* ;
- s'il n'y en a pas, recherche de *f* dans la base de l'objet *O* ;
- si cette méthode est non trouvée et appartient à une autre vue de *O*, une exception est levée.

III.4.2.7. Polymorphisme

Le polymorphisme est un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence. Ce concept n'est pas modifié par VUML. Chaque sous-classe hérite de la spécification des méthodes de ses superclasses, mais a la possibilité de modifier localement le comportement de ses opérations, afin de mieux prendre en compte les particularités liées à un niveau d'abstraction donné. Le patron proposé dans le chapitre II (section II.5.5) pour générer du code objet associé à une classe multivues et la technique proposée pour implanter l'héritage entre classes multivues garantissent que VUML supporte bien le mécanisme de polymorphisme.

III.5. Conclusion

Dans ce chapitre, nous avons décrit la sémantique statique et dynamique de VUML. Cette sémantique semi-formelle combine l'utilisation du méta-modèle, du langage naturel, et du langage formel OCL.

Les règles de bonne modélisation (well-formedness rules) décrites dans ce chapitre sont la base de la sémantique de l'outil support à VUML qui est présenté dans le chapitre IV. Ces règles sont utilisées pour vérifier les modèles VUML ce qui permet de détecter toute utilisation incorrecte des concepts de VUML.

Le fait d'avoir une sémantique semi-formelle n'est pas incompatible avec une définition correcte. Cependant, la pratique montre qu'une sémantique formelle contribue fortement à résoudre les problèmes liés aux mauvaises interprétations. Dans cette optique et à l'instar des travaux réalisés par le groupe pUML – the precise UML group – (Evans et al. 1999, Evans et al. 1998, France 1999, Bruel et al. 1998, etc) qui s'appuient sur la formalisation de UML en Z, nous projetons de réaliser une définition plus rigoureuse de la sémantique VUML. Ce travail sera abordé dans une thèse qui commencera en septembre 2005.

Chapitre IV

Outil support à VUML et application

IV.1. Introduction

Dans le chapitre précédent, nous avons présenté la sémantique de VUML. Cette sémantique est décrite en utilisant un méta-modèle, le langage naturel, et des règles de bonne modélisation (well-formedness rules) exprimées en langage formel OCL (Object Constraint Language). Dans ce chapitre nous présentons quelques aspects applicatifs de notre approche.

Dans l'objectif d'appliquer efficacement l'approche VUML, nous avons développé un outil support à VUML. Cet outil a été développé sous l'*atelier Objecteering/UML* (Objecteering-site, 2004). Le choix de cet atelier comme plate-forme de développement se justifie par le fait qu'il supporte une gestion de profils. En effet, grâce aux profils il est possible d'adapter l'outil *Objecteering/UML Modeler* (qui fait partie de l'*atelier Objecteering/UML*) pour tenir compte des contraintes de modélisation propres à des projets. Ceci permet de réduire considérablement le temps nécessaire pour réaliser des outils supportant des extensions d'UML.

Afin de mette en oeuvre l'outil support à VUML, nous avons développé deux profils :

- Le profil VUML : il permet de mener une modélisation selon VUML et de vérifier la conformité des diagrammes VUML avec la sémantique de VUML.
- Le profil « générateur de code Java » : il permet la génération du code Java à partir d'une modélisation VUML. Cette génération de code s'appuie sur le patron de génération de code présenté dans la section II.5.5 du chapitre II.

Le reste de ce chapitre est organisé comme suit : la section IV.2 donne quelques éléments sur la réalisation de l'outil support à VUML. La deuxième section VI.3 présente un exemple d'utilisation de cet outil à travers l'exemple d'un Système d'Enseignement à Distance.

IV.2. Outil support à VUML

Dans cette section, nous décrivons la réalisation de l'outil support à VUML. Nous allons tout d'abord présenter l'atelier Objecteering/UML (cf. Objecteering-site, 2004) utilisé comme environnement de développement de cette réalisation, puis nous décrirons la réalisation des modules composant l'outil support à VUML à savoir, le vérificateur de modèles et le générateur de code.

IV.2.1. Objecteering/UML

Objecteering/UML est un Atelier de Génie Logiciel (AGL) qui propose au concepteur des moyens pour décrire la solution à un problème de manière graphique et cohérente. Il fournit tous les diagrammes UML avec un contrôle de validité au cours de la modélisation. Il offre aussi la possibilité de générer du code à partir du modèle dans un langage objet cible (Java, C++,…), ainsi que d'opérer des transformations automatiques de modèle.

Objecteering/UML est le premier atelier support de la démarche MDA (Model Driven Architecture) de l'OMG (Objecteering-site, 2004). Il est structuré autour de deux outils principaux :

- *Objecteering/UML Modeler (Objecteering, 2004a)*, l'atelier de construction de modèles, paramétrable par les Profils UML,

- *Objecteering/UML Profile Builder (Objecteering, 2004b)*, l'atelier de construction de Profils UML.

Objecteering/UML fournit ainsi une séparation claire entre, d'une part, la construction d'un modèle métier indépendant des plates-formes, et d'autre part l'application sur ce modèle d'un savoir-faire technique spécifique d'une plate-forme ciblée. C'est un outil simple et puissant qui ne dépend d'aucune plate-forme. Sa simplicité est due au fait qu'il suffit de modéliser puis de sélectionner les choix techniques, pour générer l'application. En effet, Objecteering/UML propose des modèles de conception sur étagère et les générateurs dédiés aux cibles les plus diffusées. C'est un outil pérenne car il est basé sur des standards : standards de modélisation (UML pour les modèles, Profils UML pour les extensions UML, XMI pour les échanges de modèles, et MDA pour la démarche de développement), et standards de génération (Java et C++ pour les langages, composants EJB, Profils UML pour les générations). Objecteering/UML est aussi un atelier productif en démultipliant les capacités de génération par une automatisation de la génération des architectures techniques et la génération des tests.

IV.2.1.1. Principales fonctions d'Objecteering/UML

L'atelier Objecteering/UML est composé de plusieurs outils, à savoir (Objecteering-site, 2004) :

- Objecteering/UML Modeler
  - édition de modèles et diagrammes UML 1.4
  - vérification interactive de la cohérence des modèles gérés dans un référentiel unique
  - mesure de la qualité des modèles
  - génération de documentation Html et Word
  - import/export de modèles UML au format XMI et gestion du travail en groupe
- Objectering/UML Profile Builder
  - construction des profils UML
  - paramétrage d'Objecteering/UML grâce aux profils UML
  - transformation automatique de modèles
- Objectering/Java Developer
  - génération automatique de code Java
  - génération de Design Patterns pour Java

- génération des EJBs, couplés aux serveurs d'applications
- reverse-engineering de code source Java
- gestion dynamique de la cohérence code/modèle

De plus, l'atelier Objecteering/UML offre les outils suivants :

- Objecteering/C++ Developer
- Objecteering/C++ Reverse
- Objecteering/SQL Designer
- Objecteering/Corba Designer
- Objecteering/Tests for Java
- Intégration aux outils tiers

Dans la suite nous nous concentrons sur les outils *Objecteering/UML Modeler* et *Objecteering/UML Profile Builder*, car ce sont les deux outils que nous avons utilisés pour développer l'outil support à VUML.

IV.2.1.2. Objecteering/UML Modeler

Objecteering/UML Modeler est l'outil central de l'atelier Objecteering/UML. Il offre des services de création et d'édition de modèles et de diagrammes, ainsi qu'une assistance à la construction de modèles et de puissants contrôles de cohérence. La figure 72 ci-dessous présente la fenêtre principale d'Objecteering Modeler.

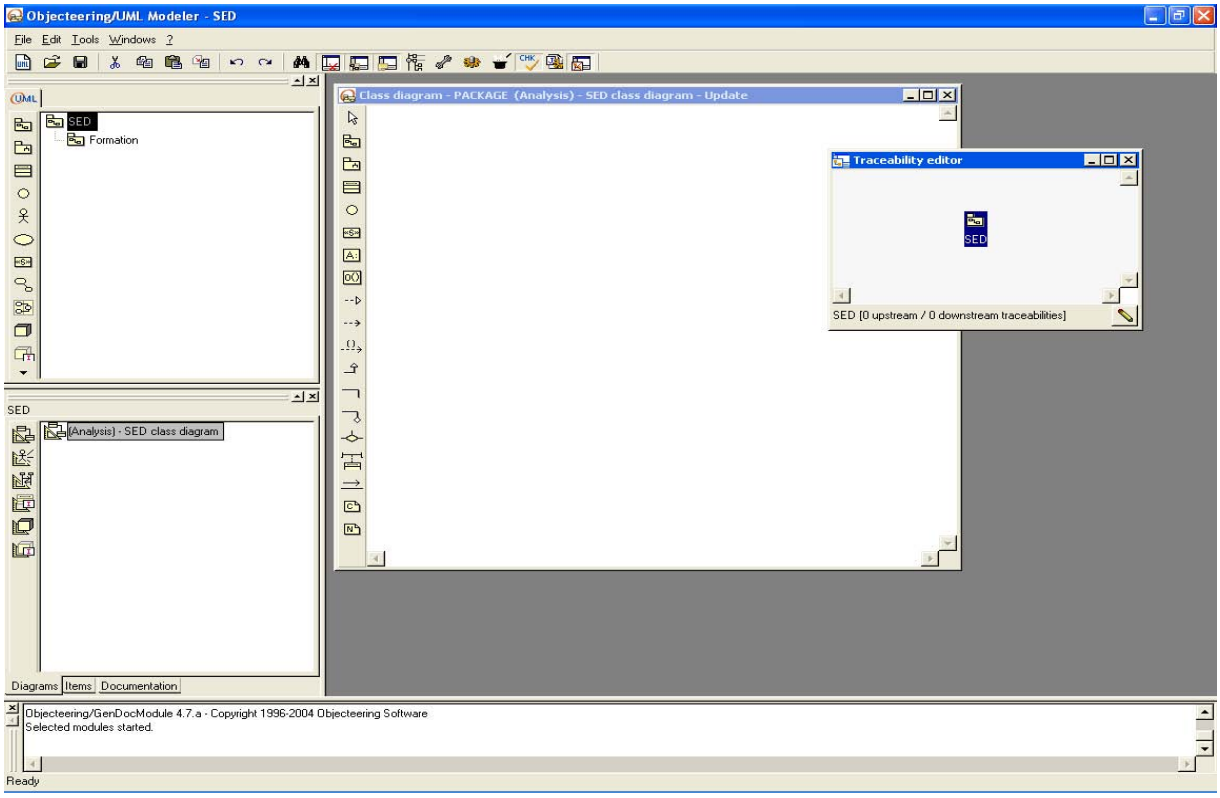

Figure 72 – *Vue générale d'Objecteering /UML Modeler sur PC*

Un des points fort de l'outil *Objecteering/UML Modeler* est qu'il peut être adapté pour tenir compte des critères de modélisation propres à des projets. Cette adaptation se fait en ajoutant/supprimant des profils. Sans la sélection des profils, l'outil *Objecteering/UML Modeler* est simplement un éditeur de modèles UML, assurant la cohérence des modèles saisis. La sélection des profils apporte tous les services spécifiques d'un domaine, tels que des extensions spécifiques du modèle UML, des générations de code adaptées, des transformations de modèle automatisant des designs patterns, etc (cf. figure 73).

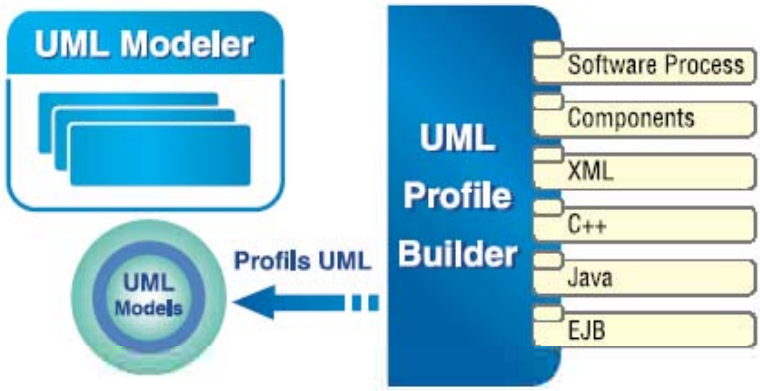

Figure 73 – *Adaptation d'Objecteering/UML Modeler par des Profils UML réalisés sous Objecteering/UML Profile Builder (figure tirée de la documentation d'Objecteering/UML)*

IV.2.1.3. Objecteering/UML Profile Builder

Alors que *Objecteering/UML Modeler* est un outil dédié aux développeurs de logiciels (analystes, concepteurs, programmeurs, etc), *Objecteering/UML Profile Builder* exploite un référentiel de profils indépendant de celui d'*Objecteering/UML Modeler* (cf. figure 74) et s'adresse à toute sorte d'utilisateur ayant un savoir-faire à définir et à faire appliquer lors d'un développement d'application UML (méthodologistes, ingénieurs processus, architectes techniques, ingénieurs qualité, etc.).

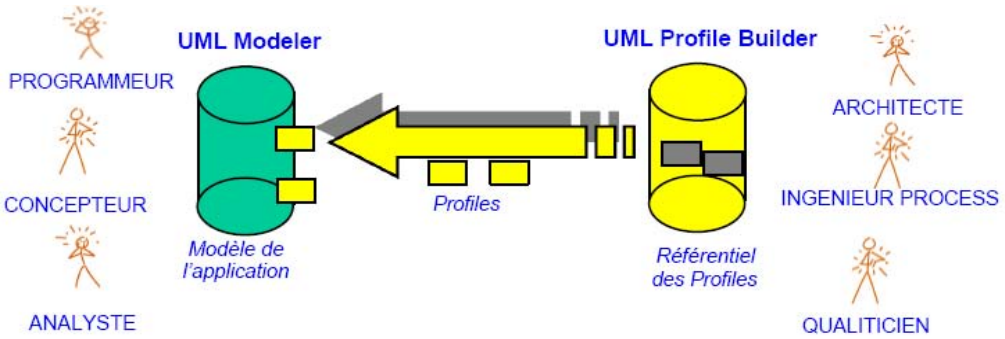

Figure 74 – *UML Modeler et UML Profile Builder ciblent des utilisateurs différents dans des référentiels différents (figure tirée de (SofTeam, 1999))*

Objecteering/UML Profile Builder est un outil de développement dédié à la construction de profils UML. Il fournit un explorateur de navigation sur le méta-modèle UML, d'accès aisé à l'utilisateur. Son langage support J, d'une syntaxe proche de Java, permet la construction de requêtes sur le méta-modèle ainsi que l'écriture de règles de transformation de modèles. De nouvelles annotations (tagged

values, stereotypes), de nouvelles rubriques de textes destinées à particulariser les modèles et guider les transformations peuvent être ajoutées au niveau du méta-modèle. La figure 75 illustre la fenêtre générale d'*Objecteering/UML Profile Builder*. Cette fenêtre montre la hiérarchie des profils de racine *default*. Un projet de profil UML est initialisé avec un certain nombre de profils UML. Ces profils ne peuvent pas être modifiés. Un profil UML est toujours créé à partir d'un profil UML parent. Le profil créé doit par la suite être "packagé" pour qu'il puisse être diffusé. Une fois la diffusion d'un profil effectuée (via l'outil *Administration* de l'atelier *Objecteering/UML*), il peut être déployé sur des projets. Chaque projet sélectionne les profils dont il a besoin, et l'atelier Objecteering/UML Modeler s'adapte dynamiquement en fonction des profils sélectionnés.

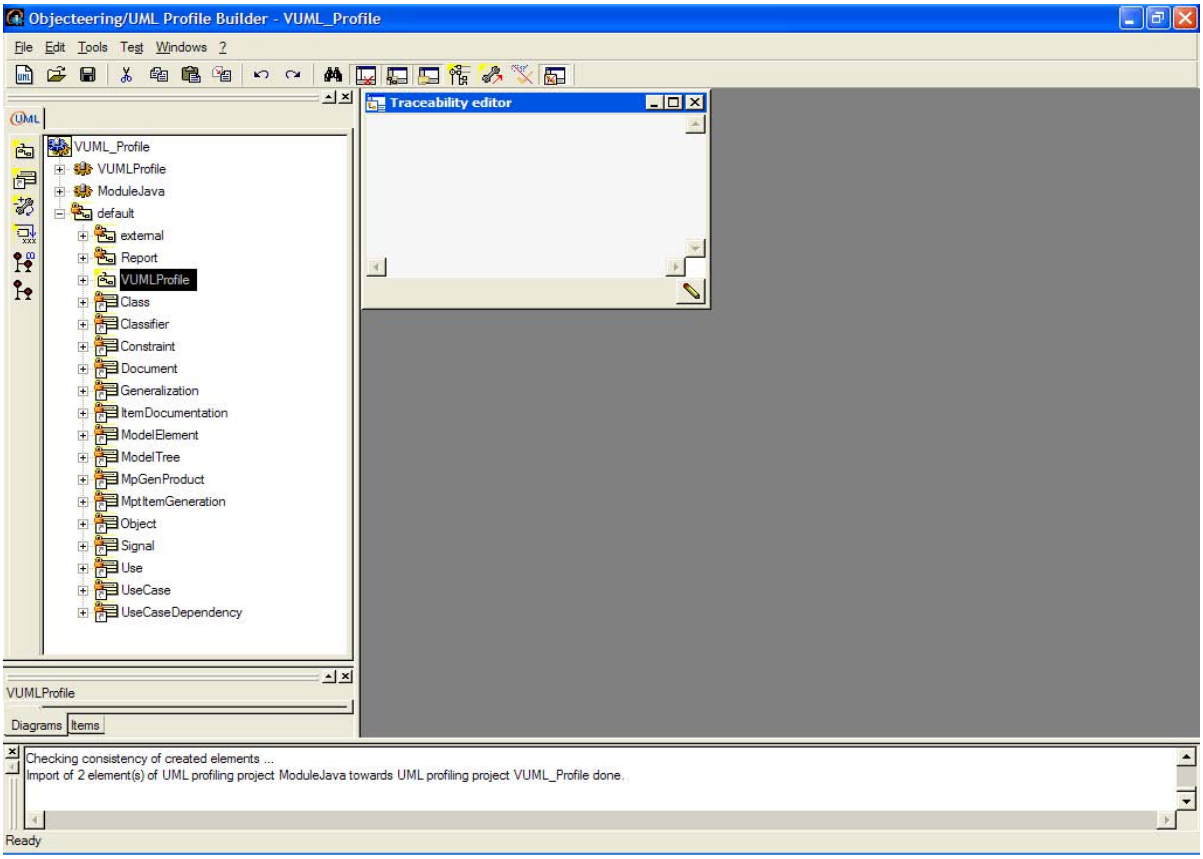

Figure 75 – *Vue générale d'Objecteering /UML Profile Builder sur PC*

IV.2.1.4. Le langage J

Le langage J (Objecteering, 2004d) est l'un des constituants essentiels de *UML Profile Builder*. C'est un langage qui permet de paramétrer et de piloter l'atelier Objecteering/UML. J est un langage objet, à la syntaxe Java, dédié à manipuler les modèles. Il est basé sur le métamodèle d'Objecteering/UML (conforme au métamodèle de UML 1.4), et permet la navigation dans des modèles UML. J est un langage interprété. Son code peut ainsi être modifié et testé rapidement. J utilise le métamodèle et manipule les éléments de modélisation créés par l'utilisateur. Par exemple, si l'utilisateur crée une classe « Personne » avec les attributs « nom » et « age », un programme J sera capable d'accéder à la classe « Personne », de chercher ses attributs et de les manipuler.

Toutes les classes utilisées dans un programme J sont prédéfinies. Elles sont soit des classes de base (int, String, …), soit des classes définies dans le métamodèle Objecteering/UML (Class, Operation, Attribute, Association, …). Le programmeur J peut déclarer de nouvelles méthodes (appelées des *méthodes J*) ou de nouveaux attributs (appelés *attributs J*) sur ces classes, mais il ne peut pas définir de nouvelles classes. Une fois qu'une méthode J est créée sur une méta-classe, son code J aura un accès direct aux propriétés de cette méta-classe. Le programmeur utilise les objets créés par les utilisateurs dans Objecteering/UML (classes, opérations, attributs, associations, …).

Comme nous l'avons déjà dit le langage J est utilisé pour naviguer dans les modèles, afin d'accéder à des informations ou d'effectuer des transformations de modèles. Cette navigation est réalisée à travers :

- *L'envoi de message* : utilisé pour appeler une méthode sur un objet.

  object.method_name(parameters) ;

- *La diffusion de message* (symbole « .< ») : utilisée pour envoyer un message à tous les objets d'un ensemble.

  referenceSet.<method_name(parameters) ;

  J utilise aussi la notion de méthode anonyme qui est un cas particulier de diffusion de message.

  ReferenceSet
  { Traitement J
  }

- *Les messages de conrôle* :
  - *Le message « select »*, qui est utilisé pour filtrer certains éléments vérifiant des critères de sélection.

    // affichage des noms des méthodes privées
    PartOperation.<select(Visibility == Private)
    { StdOut.write(Name); }

  - *Le message « while »* qui permet de répéter un traitement tant qu'une condition est vérifiée.

    // affichage des noms de méthodes différents de "display"
    PartAttribut.<while(Name != "display")
    { StdOut.write(Name); }

Le métamodèle Objecteering/UML (Objecteering, 2004c) est accessible à travers le langage J. La figure 76 présente un extrait de ce métamodèle. J utilise des méta-classes, des méta-associations, des méta-rôles, et des méta-attributs pour naviguer dans un modèle et accéder à ses informations. J permet aussi la transformation de modèles. La transformation d'un modèle s'effectue dans une *session de modification de modèle* et elle est gérée sous forme de *transaction* unitaire. Ceci offre ainsi une modification dynamique de modèle (pendant une édition de modèle sous UML modeler) contrôlée (vérifiée par les contrôles de cohérence de UML Modeler), ainsi que l'annulation et le retour en arrière sur une modification de modèle.

La figure 77 ci-dessous présente un exemple de méthode J. Cette méthode consiste à lister les noms des méthodes privées d'une classe. La figure 78 quant à elle présente une méthode qui ajoute la méthode *display()* à une classe.

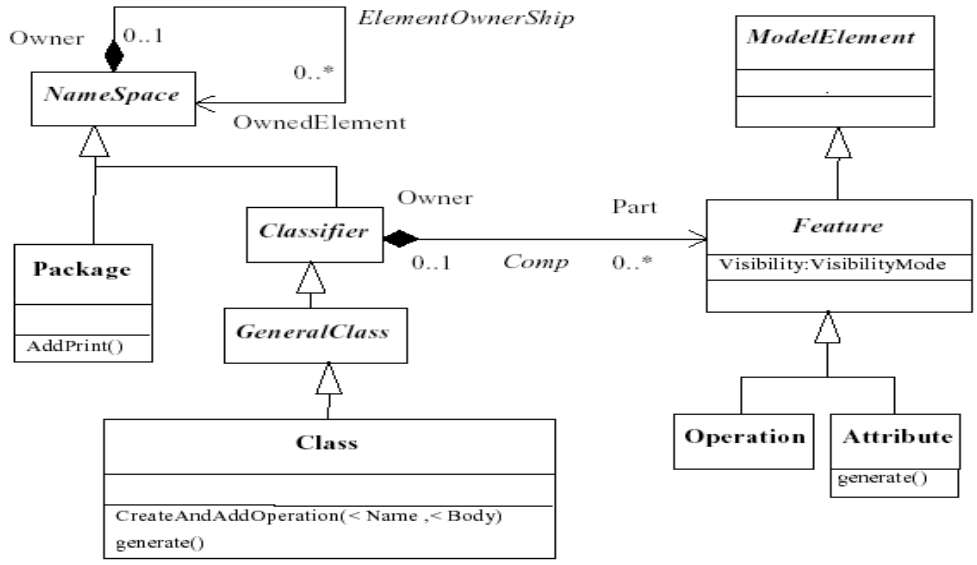

Figure 76 – *Fragment du métamodèle Objecteering/UML*

```
Class:printPrivateAttributes()   // Déclaration d'une méthode J sur la méta-classe Class
{
PartOperation    // Ensemble des méthodes de la classe
{ // Traitement sur chaque méthode
if (Visibility==Private)
StdOut.write (Name, NL);
}
}
```

Figure 77 – *Exemple d'une méthode J qui accède aux informations d'un modèle*

```
Class:addDisplay()                    // Déclaration d'une méthode J sur la méta-classe Class
{
// Déclaration d'un objet Operation
Operation M ;
// Ouverture d'une session de transformation
sessionBegin ("addDisplayOperationExample", true) ;
// création d'une méthode
M = Operation.new ();
// Affectation de "display" au nom de la méthode
M.setName ("display") ;
//Ajout de la méthode à la classe courante
this.appendPart(M) ;
//fin de la session
sessionEnd ();
//Cet ajout de méthode n'est pris en considération qu'après vérification de la cohérence du modèle
}
```

Figure 78 – *Exemple d'une méthode J qui ajoute une méthode aux classes d'un modèle*

IV.2.2. Mise en œuvre de l'outil support à VUML

Vu les caractéristiques de l'atelier Objecteering/UML tant sur le plan efficacité que sur le plan simplicité, nous avons opté pour cet atelier pour développer l'outil support à VUML. Ainsi Objecteering/UML nous a permis de réaliser le profil VUML et le profil « génération de code Java » qui permet de générer du code Java à partir d'un modèle VUML.

IV.2.2.1. Implémentation du profil VUML

Rappelons tout d'abord que le profil VUML, présenté dans le chapitre II, a pour objectif de spécialiser UML afin qu'il supporte la modélisation par vue/point de vue. L'ajout principal à UML est le concept de classe multivues. Afin d'avoir la possibilité de modéliser avec ce concept, nous avons personnalisé le métamodèle UML en introduisant un certain nombre de nouveaux stéréotypes, à savoir : *base*, *view*, *abstractView*, *viewExtension*, *viewDependency*, m*ultiViewsClass* (cf. figure 41). Ces stéréotypes sont groupés dans un profil UML. La sémantique de chacun d'entre eux est donnée dans le chapitre III consacré à la sémantique VUML. Afin de garantir le respect de cette sémantique, nous avons développé un vérificateur de modèles qui permet de vérifier les modèles VUML.

IV.2.2.1.1. Création du profil VUML

Sous *Objecteering/UML Profile Builder*, la création d'un profil se fait dans le cadre d'un projet de profil UML (UML profiling project). Ce dernier est un environnement de développement doté d'un méta-explorateur dans lequel les profils UML sont organisés hiérarchiquement. Dans chaque profil, il est possible de réaliser un ensemble d'opérations : créer un profil UML fils qui peut redéfinir des méthodes du profil UML parent, créer une référence vers une méta-classe, créer un paramètre, etc. Ainsi, le profil VUML est créé comme fils du profil prédéfini « default#external ».

IV.2.2.1.2. Création des stéréotypes

Avant de créer les différents stéréotypes introduits par le profil VUML, il faut créer des références vers les méta-classes (Class, Operation, Attribute, Association, Use, …) sur lesquelles ces stéréotypes s'appliquent. En plus des stéréotypes, il est possible dans une méta-classe de créer des méthodes J, des attributs J, des types de notes, et des types de valeurs marquées. Pour le profil VUML, nous avons créé une méta-classe Class dans laquelle nous avons défini les stéréotypes « base », « view », « abstractView », et « multiViewsClass ». Nous avons aussi créé une méta-classe « Use » (correspondant à la méta-classe *Dependency* du métamodèle UML) dans laquelle nous avons créé les stéréotypes « viewExtension » et « viewDependency » (cf. figure 79).

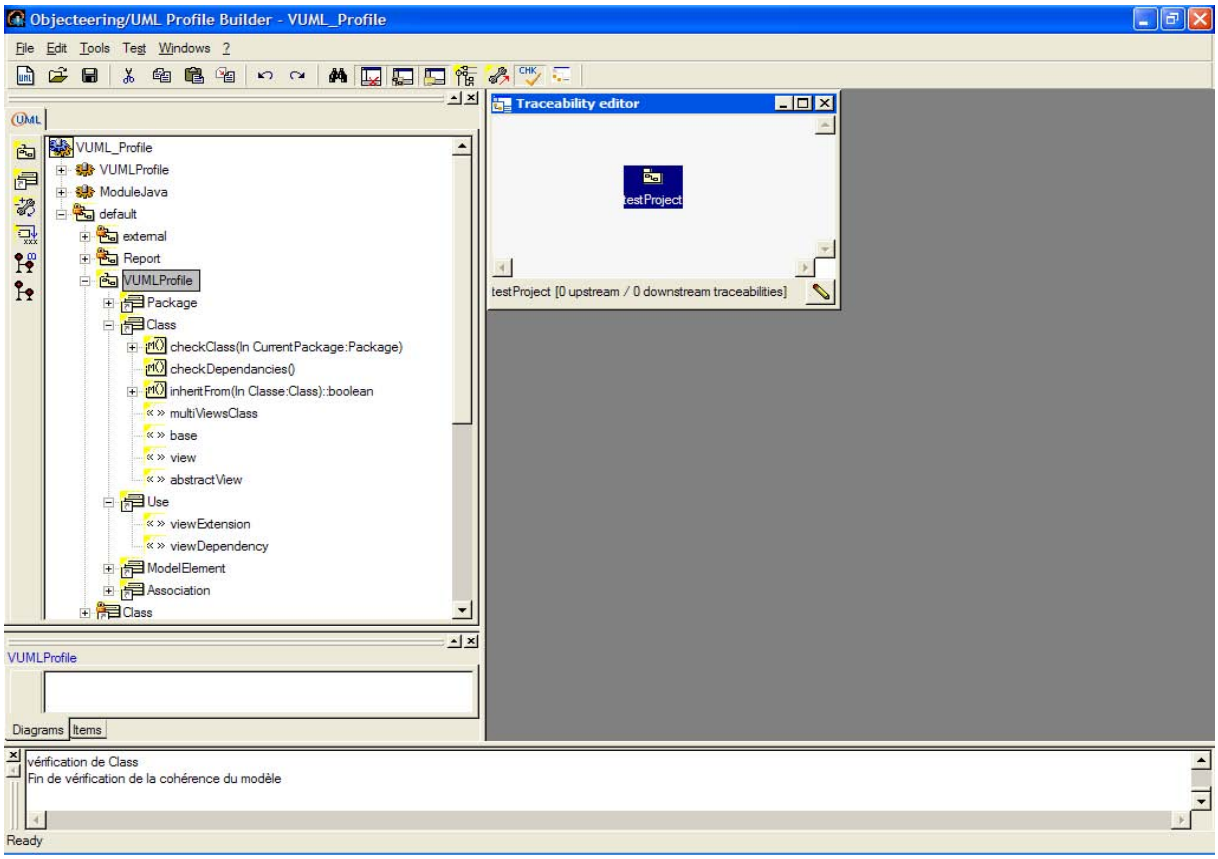

Figure 79 – *Création des stéréotypes VUML*

IV.2.2.1.3. Implémentation du vérificateur de modèles VUML

Dans cette section nous décrivons la réalisation du vérificateur de modèles VUML (design checker). Ce vérificateur implémente la sémantique VUML décrite dans le chapitre II. Cependant vu que l'atelier Objecteering/UML ne supporte pas le langage OCL, nous avons été obligés de traduire les différentes contraintes OCL en code J. L'implémentation du vérificateur de modèles VUML est faite en définissant des méthodes J sur les méta-classes concernées par la vérification. Par exemple, sur la méta-classe *Class* nous avons défini la méthode J *checkClass* qui permet de vérifier si une classe d'un modèle VUML respecte bien la sémantique VUML. La figure 80 illustre un extrait du code J de cette méthode. Ce code consiste à vérifier que toute classe descendante d'un « view » est soit un « view » soit un « abstractView ». De même, nous avons défini la méthode J *checkAssociation* sur la méta-classe *Association*. Cette méthode permet de vérifier la validité de l'utilisation des relations d'association, d'agrégation ou de composition au sein d'un modèle VUML. Un extrait du code de cette méthode est présenté dans la figure 81 ; il permet de vérifier si un « view » ou « abstractView » participe à une relation de composition en tant que composant. D'autres méthodes J sont définies sur les méta-classes *Package* et *ModelElement*. Un extrait du code J du vérificateur de modèles VUML est donné en annexe C.

```
// Un descendant direct d'un <<view>> est soit un <<view>> soit un <<abstractView>>
if ((!this.isStereotyped("view")) && (!this.isStereotyped("abstractView")))
{if (ParentGeneralization.<select(SuperTypeClass.isStereotyped("view")).size()>=1) {
StdOut.write("Erreur : la classe ", Name) ;
StdOut.write(" doit être stéréotypée soit par <<view>> soit par <<abstractViewsClass>>",NL) ;
} }
```

Figure 80 – *Code J implémentant la règle 3 portant sur le stéréotype « view » (cf. section III.4.1.1.2)*

```
String A1;
String A2;
Class C1;

If (ConnectionAssociationEnd().<select(OwnerClass.isStereotyped("view") ||
OwnerClass.isStereotyped("abstractView")).size()>=1)
{ // Récupération des classes participant à la relation
ConnectionAssociationEnd()  {
 A1 = OwnerClass.Name;
}

ConnectionAssociationEnd() {
 if (OwnerClass.Name !=A1) A2=OwnerClass.Name;
}

if (ConnectionAssociationEnd().<select(Aggregation==KindIsComposition).size()!=0)
{  // Un « view » ou « abstractView » ne peut jamais être un composant
if (ConnectionAssociationEnd().<select((OwnerClass.isStereotyped("view") ||
OwnerClass.isStereotyped("abstractView")) && (Aggregation!=KindIsComposition)).size()=1) {
StdOut.write("Erreur : la composition ",A1,"-",A2, " est interdite : Un « view » ou un « abstractView » ne peut
jamais jouer le rôle de composant",NL) ;
}
} }
```

Figure 81 – *Code J implémentant la règle 1 portant sur la relation de composition (cf. section III.4.1.2.3)*

Pour pouvoir exécuter les méthodes J d'un profil UML, il faut créer un module qui fait référence à ce profil, et sur ce module créer des commandes faisant appel aux méthodes J à exécuter. Le concept de *module Objecteering* est défini comme étant un groupe fonctionnel cohérent de profils UML et de commandes (points d'entrée pour appeler des méthodes J). Pour le profil VUML, nous avons créé le module *VUMLProfile* qui fait référence au profil *VUMLProfile*. Ce module comporte une seule commande *checkModel* qui permet d'appeler la méthode J *checkModel* définie sur la méta-classe *Package*. Cette méthode fait appel aux autres méthodes J définies dans le profil *VUMLProfile* afin de vérifier que le modèle contenu dans un package respecte bien la sémantique VUML.

IV.2.2.2. Implémentation de la génération de code

Afin de faciliter la transition d'une conception VUML vers la phase de programmation, nous avons réalisé un profil UML (génération de code) qui permet de produire du code objet à partir d'un modèle VUML. Ce générateur implémente la sémantique dynamique de VUML en appliquant le patron d'implémentation (1ère version) présenté dans la section II.5.5 du chapitre II pour générer le code cible.

Avant de décrire l'implémentation de ce profil de génération de code, nous présentons tout d'abord quelques détails concernant le patron utilisé par ce profil.

IV.2.2.2.1. Eléments sur le patron de génération de code de VUML

Comme nous l'avons déjà vu dans la section II.5.5 du chapitre II, VUML propose un patron de génération de code objet multi-cibles à partir d'une modélisation VUML. Dans la suite de cette section, nous donnons quelques détails concernant ce patron. Les exemples sont donnés en Java mais la technique proposée est valable pour tous les langages orientés objets sous réserve qu'ils supportent le mécanisme de polymorphisme. Tout d'abord, nous décrivons quelques règles qui doivent être respectées par une classe multivues. Cette dernière réalise une encapsulation totale (aucun attribut n'est accessible directement). Chaque attribut défini dans la base ou dans une vue de la classe multivues doit être déclaré comme privé et identifié par des méthodes dont le nom et la signature sont normalisés d'une manière similaire à celle utilisée par exemple pour les Beans. Nous distinguons deux types d'attributs : les attributs simples et les collections. La normalisation est la suivante : pour les attributs simples, le nom de la méthode de lecture (accesseur) d'un attribut doit obligatoirement commencer par « get » suivi par le nom de l'attribut dont la première lettre doit être en majuscule. La valeur retournée par cette méthode doit être du type de l'attribut en question. En ce qui concerne les attributs booléens, l'accesseur doit commencer par « is » au lieu de « get ». Concernant le nom de la méthode de modification d'un attribut, il doit obligatoirement commencer par « set » suivi par le nom de l'attribut dont la première lettre est en majuscule. Cette méthode ne retourne aucune valeur et doit avoir comme argument une variable du type de l'attribut. En ce qui concerne les collections, il faut aussi définir des méthodes « get » et « set » avec un argument de type entier représentant l'index de l'élément de la collection.

En plus de l'encapsulation totale qu'offre cette normalisation, elle va nous permettre de mettre en œuvre les mécanismes de mise en cohérence entre les vues et de propagation de la vue active.

IV.2.2.2.1.1. Implémentation de la gestion des vues

La gestion des vues d'une classe multivues est assurée par un gestionnaire de vues qui communique avec l'extérieur via deux interfaces (cf. figure 46 du chapitre II). La première interface (*setView()*) permet d'activer/désactiver une vue et la deuxième (*getView()*) est celle qui fournit la vue active. De plus, la gestion des vues prend en charge la mise en cohérence des vues et la propagation de la vue active. Mais avant de traiter ces points, nous devons préciser la manière dont un objet multivues implémente ses vues. La solution proposée utilise un vecteur d'objets (*_ViewsList*) pour stocker les objets vues ce qui permet d'exploiter ces objets vues en affectant l'objet représentant la vue à activer à l'attribut c*urrent_ViewExtension* (affectation valable compte tenu du mécanisme de polymorphisme). D'autre part, dans l'objectif d'optimiser la taille d'un objet multivues, la création d'un objet vue n'est faite que lors de la première activation de la vue correspondante.

Remarque : Afin de faciliter la mise en œuvre de l'héritage entre des classes multivues, l'attribut *_ViewsList* et la méthode *getView()* sont post-fixés par (_) suivi du nom de la classe multivues.

IV.2.2.2.1.1.1. L'accesseur setView()

C'est une méthode qui permet d'activer/désactiver une vue d'un objet multivues en faisant évoluer le type dynamique de l'attribut c*urrent_ViewExtension* en lui affectant l'objet représentant la vue à activer. L'opération *setView()* permet en même temps d'activer une vue et, éventuellement, de désactiver une autre vue. Cette méthode est aussi responsable de la gestion du vecteur (*_ViewsList*) contenant les vues de l'objet multivues. Elle offre aussi la possibilité de définir la base d'un objet multivues comme étant la vue active (cf. figure 82).

```
Voiture v = new Voiture();          // instanciation de Voiture
v.setView("ClientVoiture") ;        // activation de la vue ClientVoiture sur l'objet v
…
v.setView("") ;                     // activation de la base de l'objet v comme vue et désactivation de la vue ClientVoiture
…
v.setView("CommercialVoiture") ;    // activation de la vue CommercialVoiture sur l'objet v
…
```

Figure 82 – *Exemple d'instanciation d'une classe multivues et utilisation de setView()*

Cependant, l'exemple d'instanciation présenté dans la figure 82 n'est valable que si les vues ont un constructeur sans arguments. En effet, le patron d'implémentation proposé implémente les vues comme étant des classes qui peuvent éventuellement avoir des constructeurs avec arguments. Dans ce cas, un mécanisme d'initialisation s'avère nécessaire pour préciser les valeurs des arguments concernant les constructeurs des vues. Pour cela, nous proposons une solution qui exige l'exécution d'une méthode (*viewsInitiate())* permettant d'initialiser les valeurs de ces arguments avant toute activation de vue. Cette opération doit être effectuée à travers la vue d'administration, car c'est la seule vue autorisée à manipuler des données provenant d'autres vues. La figure 83 met l'accent sur ce problème en supposant que la vue *CommercialVoiture* a un constructeur avec trois arguments (*_prixConseilleCommercialVoiture*,*_prixVenteCommercialVoiture*, *_remiseCommercialVoiture*). Trois variables sont générées dans la classe *Voiture* pour stocker les valeurs de ces arguments qui seront par la suite utilisées pour créer les objets vues. La méthode *viewsInitiate()* permet de préciser les valeurs de ces arguments. La variable *_base* est une référence vers l'objet *Voiture* et la variable *_viewsInitiate* est un booléen qui sert à garder trace de l'initialisation des vues.

IV.2.2.2.1.1.2. L'accesseur *getView()*

Cet accesseur est utilisé pour accéder à la vue active (via l'attribut c*urrent_ViewExtension*). Avant de détailler le rôle de l'accesseur *getView()*, nous allons tout d'abord discuter de la manière dont un appel à une méthode est effectué. Nous avons déjà vu qu'une classe multivues *CMV* est implémentée par un ensemble de classes dont une a le même nom *CMV*. Donc, tous les objets multivues issus de la classe multivues *CMV* seront des instances de la classe *CMV* générée. Or, la question est de savoir comment traiter les invocations de méthodes sur ces objets (en particulier, les méthodes de la vue active ; par contre celles de la base ne posent pas de problème du fait que *CMV* hérite de cette base).

Pour résoudre ce problème et comme nous l'avons déjà dit dans la section II.5.4, nous avons utilisé un mécanisme d'aiguillage des appels (généré au niveau de la classe *CMV*) qui permet de rediriger l'appel vers la vue active. Pour ce faire, pour chaque méthode définie dans la classe *ViewExtension_CMV*, nous engendrons dans la classe *CMV* une autre méthode qui a la même signature et qui sert tout simplement à rediriger l'appel vers la méthode de la vue active. C'est au moment de la redirection que l'accesseur *getView()* est utilisé pour récupérer l'attribut c*urrent_ViewExtensionCMV* qui contient une référence vers la vue active.

```
class Voiture {
// Liste des variables dédiées à stocker les valeurs des arguments du constructeur de la vue CommercialVoiture
float  _prixConseilleCommercialVoiture ;
float  _prixVenteCommercialVoiture ;
float  _remiseCommercialVoiture ;
…
// Constructeurs de la base
Voiture() { … } ;
…
}
```

```
class AdminVoiture extends ViewExtensionVoiture {
// Méthode permettant d'initialiser les variables dédiées à stocker les valeurs des arguments des constructeurs des vues
viewsInitiate(float  _prixConseilleCommercialVoiture, float  _prixVenteCommercialVoiture, float  _remiseCommercialVoiture)
{
(Voiture)_base._prixConseilleCommercialVoiture = _prixConseilleCommercialVoiture ;
(Voiture)_base._prixVenteCommercialVoiture = _prixVenteCommercialVoiture ;
(Voiture)_base._remiseCommercialVoiture  = _remiseCommercialVoiture ;
(Voiture)_base._viewsInitiate = True ;
}
…
}
```

```
Voiture v = new Voiture() ;              // Instanciation de Voiture
v.setView("CommercialVoiture ") ;    // Erreur : la vue CommercialVoiture n'est pas encore initialisée
v.setView("AdminVoiture") ;           // activation de la vue AdminVoiture
v.viewsInitiate(0,0,0) ;                 // Initialisation des vues (précision des valeurs des arguments des constructeurs des vues)
v.setView("CommercialVoiture ") ;    // Activation de la vue CommercialVoiture
…
v.setView("ClientVoiture") ;             // Activation de la vue ClientVoiture
…
```

Figure 83 – *Illustration du rôle de la méthode viewsInitiate()*

Supposons que la classe multivues *CMV* ait une vue *Vuek* qui contient une méthode *f()* ayant comme type de valeur de retour *T* (qui peut être *void*) et comme liste d'arguments *Args* (qui peut être vide). Cette méthode peut être une redéfinition d'une méthode de la base de la classe *CMV*. La figure 84 illustre le mécanisme d'aiguillage généré pour cette méthode. Dans la méthode représentant ce mécanisme d'aiguillage (la méthode *f()* générée dans la classe *CMV*) il y a une instruction qui permet

d'appeler la méthode *f()* sur la vue active (si la *vuek* n'est pas active, c'est la méthode *f()* de la classe *ViewExtension_CMV* qui sera appelée). Il faut noter que si cette méthode retourne une valeur, ceci sera pris en considération au moment de la redirection. Un exemple concret illustrant le mécanisme d'aiguillage des appels est présenté sur les figures 85 et 86.

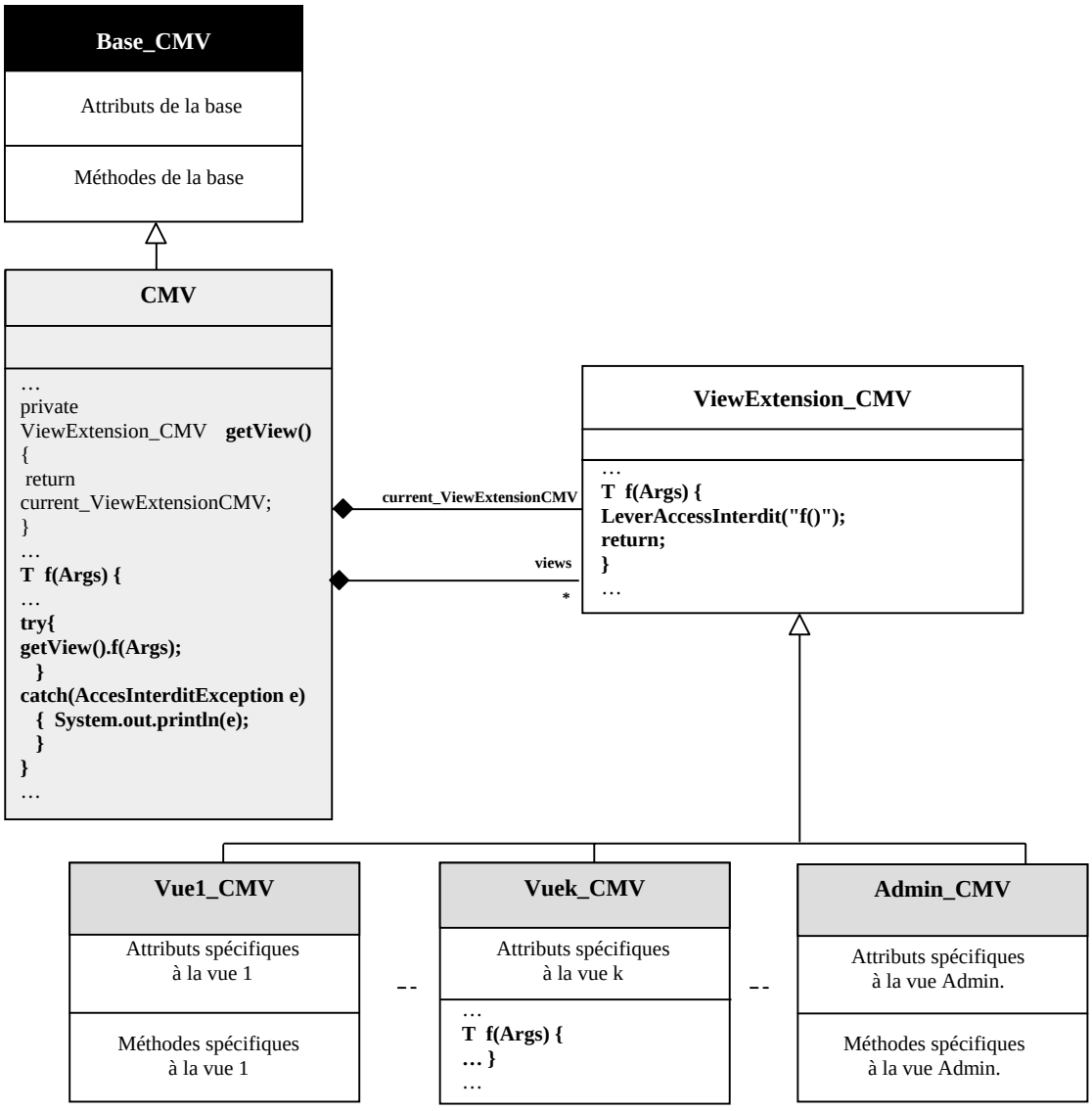

Figure 84 – *Illustration du mécanisme d'aiguillage des appels et utilisation de getView()*

```
class Base_Voiture
{
 private String ref;
 private String marque;
 …
public void afficherInfos(){
 …}
… }
```

```
class  Voiture extends Base_Voiture {

// Constructeur
public Voiture() {
current_ViewExtensionVoiture= new  ViewExtension_Voiture(this);
…
}

// Partie concernant la gestion des vues
private
ViewExtension_Voiture current_ViewExtensionVoiture ;

// Vecteur des vues
 private Vector   _ViewsList_Voiture = new Vector();
…
// Vue active
 protected String   _ActiveView = new String("");

// getView_Voiture : méthode d'accès à la vue active
 protected ViewExtension_Voiture  getView_Voiture() {
 return current_ViewExtensionVoiture;
 }
// setView : méthode d'activation de vues
 public void setView(String V) { ... }
```

```
// Mécanismes d'aiguillage
int  _i = 1;   // utilisé pour ne pas avoir des appels cycliques lors
               // de l'appel d'une méthode de la base redéfinie
               // dans une vue.

public void afficherInfos() {
if (_ActiveView.equals("ClientVoiture"))
  { try {   getView_Voiture().afficherInfos(); }
    catch (AccesInterditException e)  {
       System.out.println(e);   }
  }
else {
if (_i==2) { _i=1;  super.afficherInfos(); }
else {
if ((_ActiveView.equals("CommercialVoiture")) ||
(_ActiveView.equals("MecanicienVoiture")) ||
(_ActiveView.equals("ElectricienVoiture")) ||
(_ActiveView.equals("CarrossierVoiture")))
  { _i++;
     try {  getView_Voiture().afficherInfos();  }
    catch (AccesInterditException e) {
       System.out.println(e); }
  }
else  super.afficherInfos();
} }


public void modifierInfos() {
try {  getView_Voiture().modifierInfos(); }
catch(AccesInterditException e) {
 System.out.println(e); }
}
… }
```

Figure 85 – *Un extrait du code Java généré pour la classe multivues Voiture*

```
Voiture v = new Voiture() ;          // Instanciation de Voiture
v.setView("ClientVoiture") ;         // activation de la vue ClientVoiture
v.afficherInfos() ;                  // redirection de l'appel vers la méthode afficherInfos() de la vue ClientVoiture
v.modifierInfos() ;                  // lève une exception car la vue ClientVoiture ne donne pas accès à la méthode modifierInfos()
…
v.setView("CommercialVoiture") ;     // activation de la vue CommercialVoiture
v.afficherInfos() ;                  // redirection de l'appel vers la méthode afficherInfos() de la vue CommercialVoiture
v.modifierInfos() ;                  // redirection de l'appel vers la méthode modifierInfos() de la vue CommercialVoiture
v.repondreProposition() ;            // redirection de l'appel vers la méthode repondreProposition() de la vue CommercialVoiture
…
v.setView("");                       // activation de la base de l'objet v
v.afficherInfos() ;                  // redirection de l'appel vers la méthode afficherInfos() de la base
v.reparerPanne() ;                   // lève une exception car la base ne donne pas accès à la méthode reparerPanne()
…
v.setView("MecanicienVoiture") ;     // activation de la vue MecanicienVoiture
v.reparerPanne() ;                   // redirection de l'appel vers la méthode reparerPanne() de la vue MecanicienVoiture
```

Figure 86 – *Exemple d'appels en Java sur un objet de type Voiture*

IV.2.2.2.1.1.3. Implémentation de la mise en cohérence des vues

Afin de mettre en oeuvre le mécanisme de mise en cohérence des vues (cf. section II.3.5 du chapitre II), nous avons utilisé une solution qui consiste à intégrer ce mécanisme dans l'accesseur *setView()* ce qui permet de répercuter les modifications entre les vues dépendantes à chaque changement de point de vue. Ceci est possible car la classe centrale générée (qui porte le même nom que la classe multivues concernée) est dotée d'une liste d'objets (*_ViewsList*) représentant les différentes vues. Cependant, l'inconvénient de cette solution est que le modèle reste incohérent jusqu'à l'exécution du prochain changement de point de vue. Par conséquent, cette solution est adaptée à un système mono-utilsateur et non pas aux systèmes multi-utilisateurs où plusieurs vues peuvent être actives simultanément. Une deuxième solution consisterait à implémenter ce mécanisme d'une manière distribuée dans les différentes vues qui sont en dépendance (possédant des données dépendantes). Pour ce faire, on peut profiter du fait qu'il y a une encapsulation totale des informations stockées dans les vues, c'est-à-dire que l'accès à un attribut passe toujours par l'accesseur correspondant. L'idée est donc d'intégrer la dépendance concernant chaque attribut dans son accesseur. Toutefois, cette manière de faire possède 2 inconvénients gênants : d'une part, cette solution exige d'avoir des accès entre les vues ce qui n'est pas conforme à la définition d'une vue ; d'autre part, le mécanisme de mise en cohérence est appelé même s'il n'y a pas d'incohérences dans le modèle ce qui entraîne une dégradation des performances du système. Pour ces raisons, nous avons opté pour la première solution.

La figure 87 présente un extrait de la classe générée *Voiture*. Elle met l'accent sur la manière dont la mise en cohérence des vues est réalisée. La méthode g*etViewsList_Voiture()* représente l'accesseur de la liste des objets correspondant aux vues de la classe *Voiture*. La variable *_ViewsNamesList_Voiture* représente la liste des noms de ces vues ; tandis que les variables *indexClient* et *indexCommercial* sont, respectivement, les index des objets représentant les vues *ClientVoiture* et *CommercialVoiture* dans la liste *_ViewsList_Voiture*. Des opérateurs de cast sont nécessaires pour convertir les objets de type *Object*. Le mécanisme de mise en cohérence entre les vues est matérialisé par la méthode *viewsUpdate()* qui est appelée par *setView()* avant de changer le point de vue. La méthode *viewsUpdate()* permet de réaliser les répercussions des modifications entre les vues en dépendance. Pour ce faire, elle utilise la liste des objets représentant les vues dans la classe *Voiture* (*_ViewsList_Voiture*). Puisqu'une vue n'est créée que lors de sa première activation, la méthode *viewsUpdate()* vérifie l'existence d'une vue avant de faire la répercussion des modifications de cette vue vers d'autres vues. Le code de la figure 87 implémente les dépendances illustrées dans la figure 39 du chapitre II entre les vues *ClientVoiture* et *CommercialVoiture*. Rappelons que l'une de ces dépendances exprime que l'ensemble des propositions de prix reçues par le commercial relativement à une voiture inclut l'ensemble des propositions de prix effectuées par un client pour acheter cette voiture ; tandis que l'autre dépendance spécifie que le commercial et le client doivent utiliser les mêmes valeurs pour le prix de vente et la remise.

```
class  Voiture  extends  Base_Voiture {
…
// Vecteur des vues
 private Vector   _ViewsList_Voiture = new Vector();

// Vecteur des noms des vues
 private Vector   _ViewsNamesList_Voiture = new Vector();

// Vue active
 protected String   _ActiveView=new String("");

// setView : méthode d'activation de vues
 public void  setView(String V) {
…
viewsUpdate(V);                    // appel au mécanisme de mise en cohérence entre les vues
…
}

// Mécanisme de mise en cohérence entre les vues
private void viewsUpdate(String V) {

// Mise en cohérence de la vue ClientVoiture
if (V.equals("ClientVoiture"))
{
 if (ViewsNamesList_Voiture.contains("CommercialVoiture"))
{ ((ClientVoiture)getViewsList_Voiture(indexClient)).setPrixVente(((CommercialVoiture)getViewsList_Voiture(indexCommercial)).getPrixVente());
   ((ClientVoiture)getViewsList_Voiture(indexClient)).setRemise(((CommercialVoiture)getViewsList_Voiture(indexCommercial)).getRemise());
}
}

// Mise en cohérence de la vue CommercialVoiture
if (V.equals("CommercialVoiture"))
{
 if (ViewsNamesList_Voiture.contains("ClientVoiture"))
{ ((CommercialVoiture)getViewsList_Voiture(indexCommercial)).getProppositionsDAchat().addAll(
            ( (ClientVoiture)getViewsList_Voiture(indexClient)).getProppositionsDAchat () );
 }
}
}
…
}
```

Figure 87 – *Extrait de la classe Voiture illustrant l'implémentation du mécanisme de mise en cohérence des vues*

IV.2.2.2.1.1.4. Implémentation de la propagation de la vue active

Le mécanisme de propagation de la vue active consiste à activer la vue active d'un objet multivues - instance d'une classe multivues - sur les objets qui lui sont reliés par des liens d'association, d'agrégation ou de composition (cf. section II.5.3 du chapitre II). Une première solution concernant l'implémentation de ce mécanisme est d'utiliser l'accesseur *setView()* de la classe multivues pour faire appel aux autres accesseurs *setView()* des attributs multivues de cette classe. Cependant, ceci n'est pas adapté pour traiter les collections dont les éléments peuvent être des objets de types différents. La deuxième solution est d'intégrer un appel à l'accesseur *setView()* (avec comme argument la vue active) dans les accesseurs des attributs multivues de la classe (ce qui va activer la vue active de l'objet multivues sur ces attributs) avant l'instruction *return* (qui permet de retourner la valeur d'un attribut). Nous avons opté pour cette solution car le seul inconvénient qu'elle possède est que le mécanisme de propagation de la vue active est appelé même si la vue en question est déjà active.

La classe multivues *Voiture* est reliée par une agrégation à la classe multivues *Constructeur* et par une association à une collection (*accidents*) dont le type des éléments est la classe multivues *Accident* (cf. figure 47, chapitre II). Or, une vue activée sur un objet de type *Voiture* doit être propagée vers ses deux composants - *constructeur* et *accidents* - (cf. section II.5.3). La figure 88 montre un extrait de la classe *Voiture* qui met l'accent sur le rôle des accesseurs relatifs aux attributs représentant ces composants. Ces accesseurs sont dotés d'une instruction qui active la vue active sur les attributs associés avant de retourner leurs valeurs (si la vue active n'est pas une vue de la classe multivues d'un attribut, l'activation ne sera pas faite). La vue active est supposée stockée dans la variable *_ActiveView*.

```
class Base_Voiture  {
…
private Constructeur constructeur = new Constructeur() ;    // instanciation d'un attribut multivues de type Constructeur
private Vector accidents = new Vector();                    // instanciation d'un vecteur qui sera utilisé pour stocker des
                                                            // objets multivues de type Accident
…
}
```

```
class Voiture extends Base_Voiture  {
…
// Redéfinition de l'accesseur de l'attribut constructeur
public Constructeur getConstructeur() {
 // Propagation de la vue active : il faut tester si l'attribut est doté de cette vue
if ((super.getConstructeur().getViewsNamesList_Constructeur().contains(_ActiveView))
   super.getConstructeur ().setView(_ActiveView);
   return super.getConstructeur();
}
…
// Redéfinition de l'accesseur de l'élément d'index i dans le vecteur accidents
public Accident getAccidents(int i) {

if (((Accident)(super.getAccidents(i)).getViewsNamesList_Accident().contains(_ActiveView))
                    ((Accident)(super.getAccidents(i))).setView(_ActiveView);
return (Accident)(super.getAccidents(i));
}
… }
```

Figure 88 – *Extrait de la classe Voiture illustrant l'implémentation du mécanisme de propagation de la vue active*

IV.2.2.2.1.2. Implémentation de l'héritage entre classes multivues

Comme nous l'avons déjà dit (cf. section II.3.4 du chapitre II), le mécanisme d'héritage n'est pas modifié par l'introduction de la notion de classe multivues. Dans cette section, nous présentons la manière dont nous avons implémenté l'héritage entre deux classes multivues (les classes ordinaires sont considérées comme des classes multivues constituées seulement de la base). Rappelons que l'héritage entre une classe multivues *CMV2* et une classe multivues *CMV1* se fait en déclarant un lien d'héritage dans la base de *CMV2* vers la base de *CMV1*, et éventuellement, un lien d'héritage entre les

vues s'il y a une redéfinition de vues. Le code implémentant ces deux classes concentre toutes les ressources dans les classes (générées) portant leurs noms. Donc, il suffit que l'héritage se fasse entre la classe générée *Base_CMV2* et la classe *CMV1* (pour ne pas avoir de l'héritage multiple au niveau de la classe fille *CMV2* car elle hérite déjà de la classe *Base_CMV2*). Cette manière de faire permet de disposer de toutes les ressources de la classe *CMV1* au niveau de la classe *CMV2*. Le problème qui reste à régler est l'activation/désactivation des vues de la classe descendante. Notre solution consiste à prendre en compte cette tâche au niveau de l'accesseur *setView()* de la classe descendante *CMV2*. Ainsi, pour activer une vue *V*, l'accesseur *setView()* de *CMV2* doit :

- activer la vue *V* si elle est une vue propre à *CMV2*,
- déléguer l'activation de la vue *V* à l'accesseur *setView()* de la classe parente *CMV1* si *V* est une vue de cette classe,
- si *V* est une vue de *CMV1* redéfinie dans *CMV2* alors l'accesseur *setView()* de la classe *CMV2* doit activer la vue *V* et appeler l'accesseur *setView()* de la classe parente *CMV1* pour activer cette même vue.

La figure 89 présente un extrait du pseudo-code de l'accesseur *setView()* de la classe multivues *VoitureCourse* qui hérite de la classe multivues *Voiture* (cf. figure 37, chapitre II). L'activation de la vue *CommissaireVoitureCourse* (propre à la classe *VoitureCourse*) ne va pas entraîner l'appel de l'accesseur *setView()* de la classe parente *Voiture* ; en effet, cette vue n'est pas une vue de la classe *Voiture*. Par contre, l'activation de la vue du mécanicien *MécanicienVoitureCourse* (redéfinie au niveau de la classe *VoitureCourse*) est traitée par les accesseurs *setView()* de la classe descendante et de la classe parente. Une vue de la classe *Voiture* non redéfinie dans la classe *VoitureCourse* (*ClientVoiture* par exemple) est activée par le *setView()* de la classe *Voiture*.

```
public setView(String V) {
…
if ((getViewsNamesList_VoitureCourse.contains(V))          // getViewsNamesList_VoitureCourse() et
    { Traitement pour affecter l'objet représentant la vue V à   // getViewsNamesList_Voiture() sont, respectivement,
      current_ViewExtensionVoitureCourse}                  // les accesseurs de la liste des noms des vues de la classe
                                                           // VoitureCourse et de la classe Voiture
if ((getViewsNamesList_Voiture().contains(V))
  super.setView(V);                                        // appel du modificateur setView() de la classe parente Voiture
}
```

Figure 89 – *Extrait du pseudo-code du modificateur setView() de la classe multivues VoitureCourse*

Remarque : Pour les langages qui supportent l'héritage multiple, il est possible qu'une classe multivues hérite de plusieurs classes multivues. Dans ce cas, la classe héritière doit prévoir la résolution des conflits entre les différents accesseurs *setView()* des différentes classes parentes. Ceci peut être réalisé par exemple dans le langage Eiffel par des renommages.

IV.2.2.2.2. Implémentation du profil de génération de code Java

Afin d'assister le développeur VUML pendant la phase d'implémentation, nous avons décidé d'automatiser la génération du code en utilisant le patron de génération de code de VUML. Cette automatisation est réalisée sous forme d'un profil UML permettant la génération de code directement à partir d'un modèle VUML. Ce profil est créé comme fils du profil prédéfini « default#external#Code » afin qu'il hérite les propriétés générales d'un générateur de code.

Dans un premier temps, nous avons choisi de cibler le langage Java mais les techniques utilisées dans cette génération de code sont réutilisables pour cibler d'autres langages Objet. Un extrait du code J du générateur de code Java réalisé est présenté dans l'annexe C.

Pour des raisons purement liées à l'implémentation, nous avons décidé d'ajouter un nouveau stéréotype « redefined » qui s'applique à l'élément de modélisation *Operation*. Ce stéréotype est utilisé pour designer les méthodes des vues qui redéfinissent une méthode de la base d'une classe multivues et qui font appel à la méthode redéfinie. Cette manière de faire permet, lors du parcours du modèle VUML, de détecter ces méthodes et de générer un code Java évitant les appels cycliques entre la base et la vue concernée contenant ce genre de méthodes.

Avec ce générateur de code Java, la phase de programmation est réduite à l'essentiel. Une fois le code généré, il ne reste qu'à compléter le code algorithmique des méthodes pour avoir le code complet du système. Le code produit par ce générateur répond aux critères de lisibilité, de maintenabilité, de robustesse et d'efficacité. En plus de la génération de code, le profil réalisé permet de visualiser et d'éditer le code généré en utilisant des éditeurs externes.

IV.3. Application

Cette section a pour objet de présenter l'application de VUML en utilisant son outil support. Pour illustrer cette application, nous nous appuyons sur la modélisation d'un Système d'Enseignement à Distance (SED par la suite) tel qu'il en existe dans les universités. Une présentation plus détaillée de cet exemple peut être trouvée dans (Nassar et al., 2002). Cet exemple a été choisi comme étude de cas commune par les partenaires du réseau STIC franco-marocain en Génie Logiciel (Coulette et al., 2002). Le cahier des charges de ce système est présenté dans l'annexe D.

IV.3.1. Présentation de l'application

Le Système d'Enseignement à Distance (SED) permet à des étudiants de suivre des formations à distance. Pour simplifier, nous considérons que le système ne cible qu'un seul site, géré par un responsable. Un site propose des formations (cours) auxquelles peuvent s'inscrire des personnes (alors qualifiées d'étudiants), qui acquittent des droits en fonction des formations suivies. Afin de réduire la taille des modèles nous nous limitons aux acteurs et aux activités suivants :

- les étudiants s'inscrivent à des formations, suivent des formations à distance, et posent des questions sur les formations.
- les enseignants auteurs de formations produisent des formations et les mettent à jour, produisent les compléments de formation et la liste des ouvrages conseillés pour approfondir une formation, …
- les enseignants tuteurs assurent le suivi des étudiants, répondent à leurs interrogations, posent et corrigent des examens, …
- le responsable de site ajoute de nouvelles formations, gère les inscriptions des étudiants et affecte les enseignants responsables des formations.
- Le conseil pédagogique (doté d'un président) réunit les jurys, fait le bilan pédagogique, définit la stratégie du SED, traite les litiges et les cas particuliers, …
- le directeur du SED définit la stratégie globale et produit les bilans de fonctionnement.

La figure 90 illustre un sous-ensemble des cas d'utilisation identifiés : Gestion de la scolarité, Gestion des formations, Suivi d'une formation, Supervision.

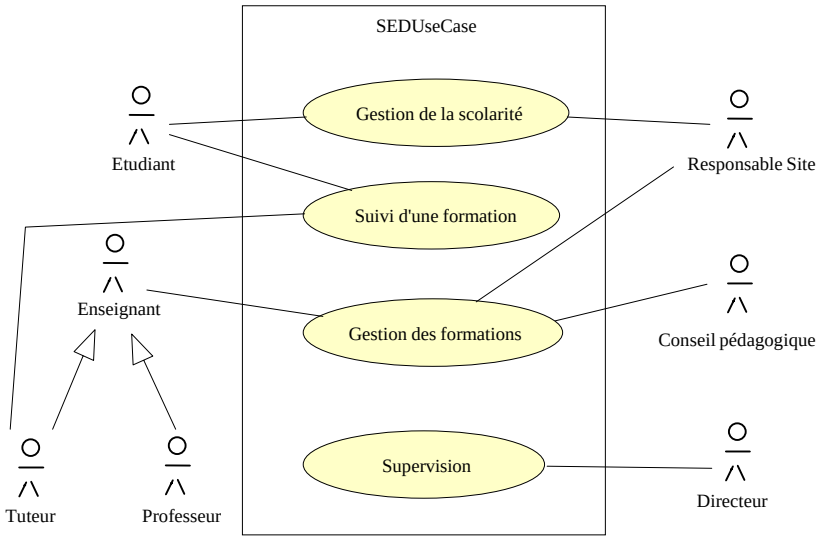

Figure 90 – *Cas d'utilisation du SED (exemple simplifié)*

IV.3.2. Modèle VUML de l'application

La figure 91 ci-dessous illustre un extrait du diagramme VUML du SED. Pour des raisons de lisibilité, certaines classes multivues ne sont pas éclatées (classes stéréotypées par « multiViewsClass »). Cette figure montre aussi qu'il y a une dépendance entre la vue de l'enseignant et la vue du responsable du site. Cette dépendance exprime le fait qu'un enseignant accède aux informations des mêmes étudiants que ceux inscrits à son cours auprès du responsable du site.

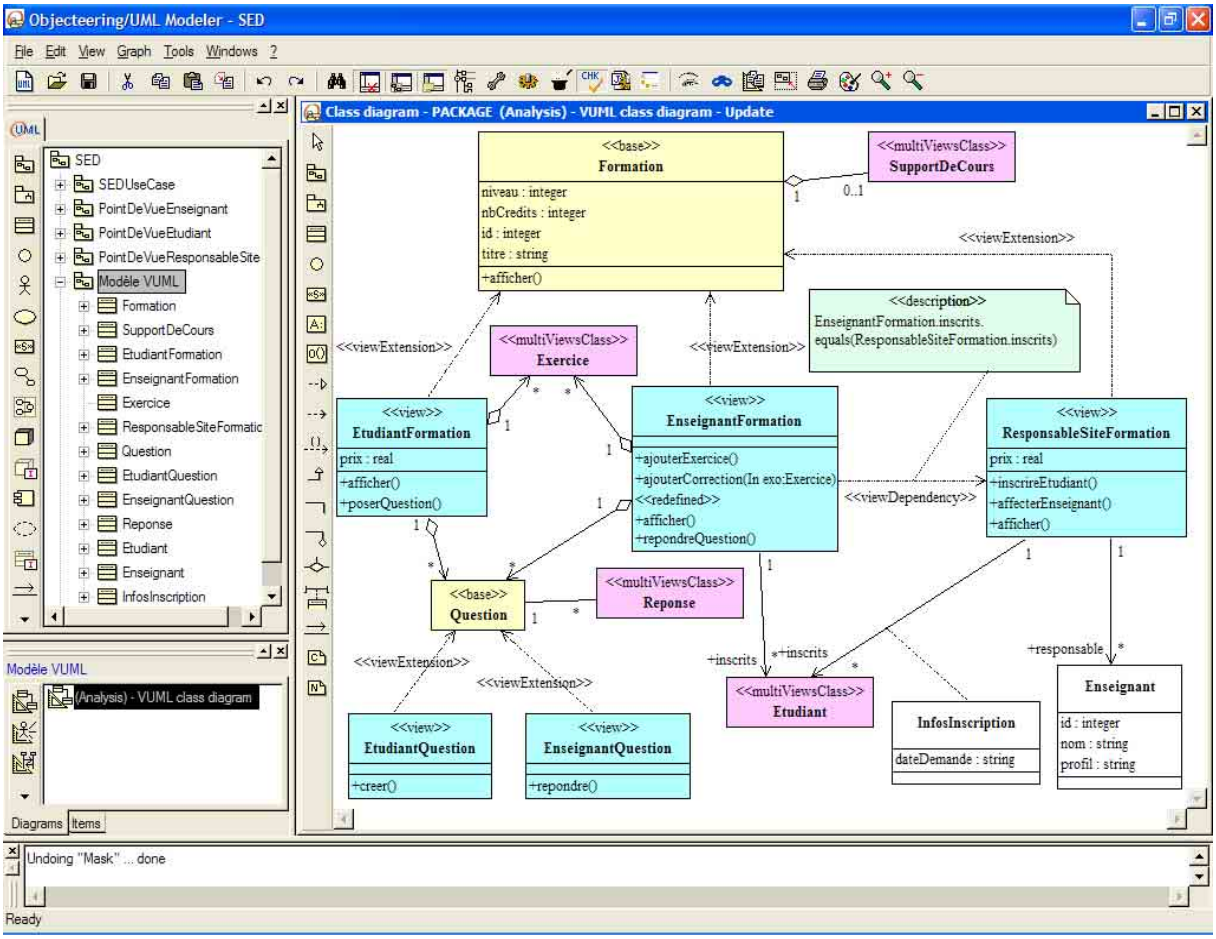

Figure 91 – *Diagramme VUML avec les classes multivues Formation et Question*

IV.3.3. Vérification du modèle VUML et génération de code

Une fois le diagramme VUML élaboré, il doit être vérifié afin de s'assurer qu'il respecte bien les règles de bonne modélisation définies dans VUML. La figure 92 ci-après présente deux exemples d'erreurs détectées par le vérificateur de l'outil support à VUML. Le premier exemple est une erreur qui concerne la relation d'agrégation entre la classe *Question* et la vue *EnseignantFormation* qui doit être navigable uniquement dans le sens *EnseignantFormation-Question*. Le deuxième exemple est une erreur liée au non respect du fait que la source d'une dépendance « viewExtension » doit être un « view » ou « abstractView ».

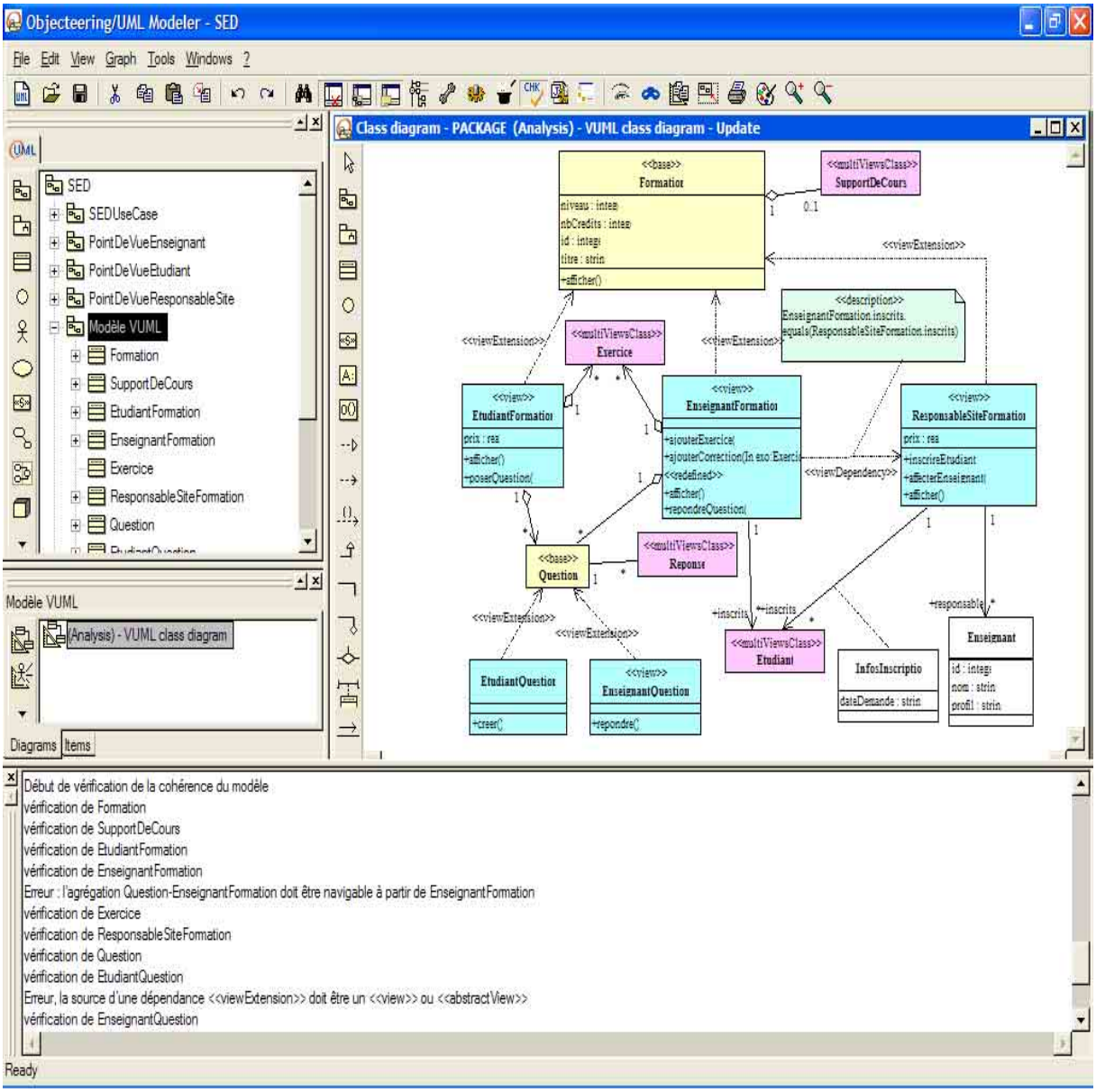

Figure 92 – *Exemples d'erreurs détectées lors de la vérification d'un modèle VUML*

Quand les erreurs détectées par le vérificateur de modèles VUML sont toutes corrigées, nous procédons à la génération de code à l'aide du générateur de code Java offert par l'outil support à VUML (cf. figure 93). Le code généré est présenté dans l'annexe E.

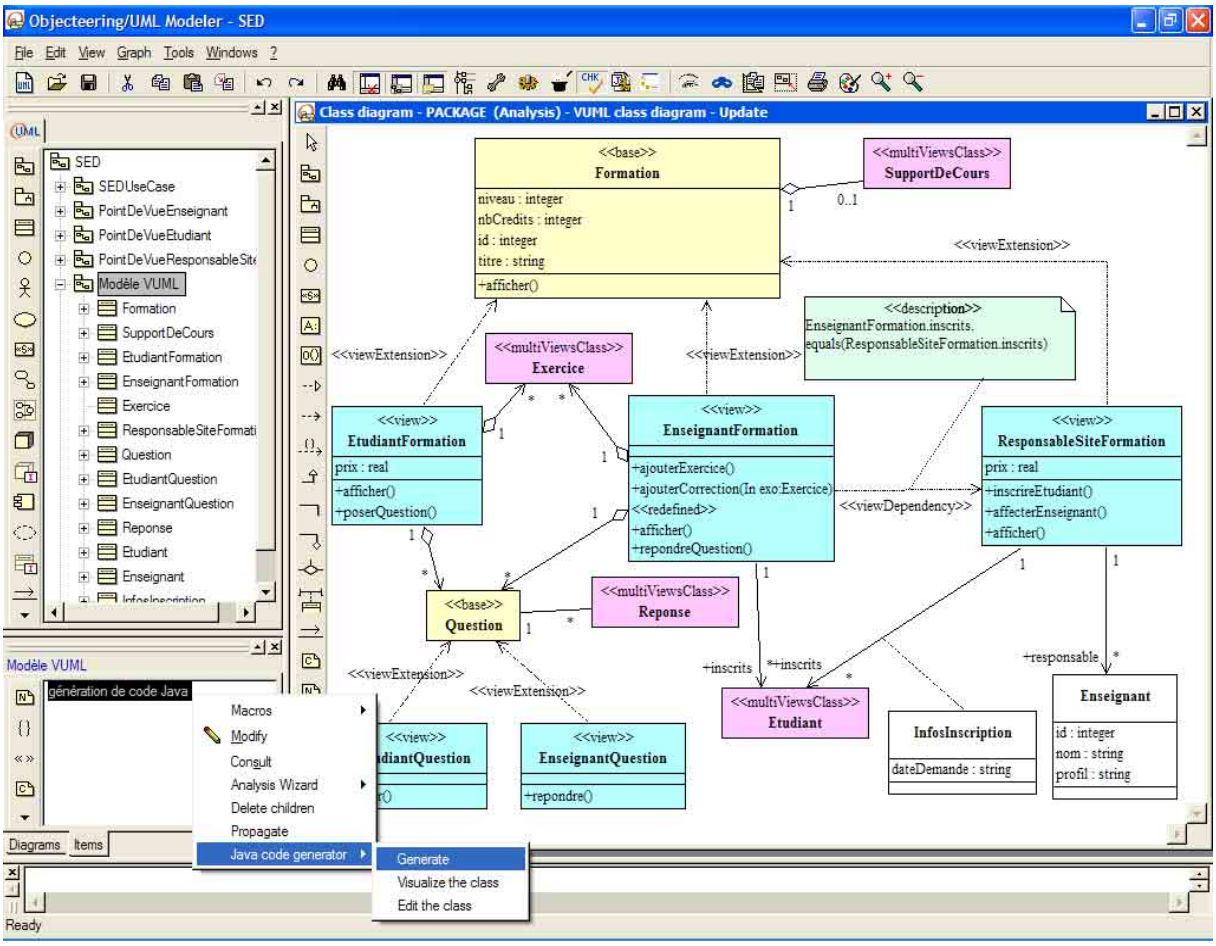

Figure 93 – *Génération du code Java du SED*

IV.4. Conclusion

Ce chapitre a décrit la réalisation de l'outil support à VUML ainsi que l'expérimentation de cet outil à travers un exemple d'un système d'enseignement à distance. Cette réalisation a profité des avantages offerts par l'atelier *Objecteering/UML* et surtout ses modules *Objecteering /UML Profile Builder* et *Objecteering/UML Modeler*. De plus, l'atelier *Objecteering/UML* nous a permis de réduire considérablement le temps nécessaire pour mettre en œuvre l'outil support à VUML.

En plus des fonctionnalités de l'outil *Objecteering/UML Modeler*, l'outil support à VUML offre deux autres fonctionnalités. La première concerne la vérification des modèles VUML. Cette vérification supporte toute la sémantique VUML statique décrite dans le chapitre III. La deuxième est la génération de code correspondant à un modèle VUML. Cette génération de code implémente le patron d'implémentation présenté dans le chapitre II. Elle génère le code correspondant à tous les concepts et mécanismes VUML sauf les dépendances fonctionnelles et la propagation de la vue active qui sont en cours d'implémentation. Pour l'instant la génération de code cible le langage Java mais les techniques employées sont réutilisables pour cibler d'autres langages objets.

Chapitre V

Démarche de mise en œuvre de VUML dirigée par les modèles

V.1. Introduction

Beaucoup pensent que l'ingénierie logicielle dirigée par les modèles est l'avenir du génie logiciel. Les grands acteurs du domaine du logiciel (Bill Gates (MicroSoft), Richard Soley (Directeur de l'OMG), Grady Booch, Jim Rumbaugh, Ivar Jacobson (concepteurs d'UML),…) affirment que les modèles doivent être au centre du cycle de vie des applications (Blanc, 2005).

Dans ce contexte, l'OMG (Object Management Group) a proposé, en novembre 2000, l'approche Model Driven Architecture (Soley et al., 2000). Cette approche a, initialement, visé de s'appuyer sur UML pour décrire séparément les spécifications fonctionnelles d'un système et les spécifications de son implémentation sur une plate-forme donnée. L'approche MDA est basée sur la production de modèles indépendants des plates-formes (PIM ou Platform Independent Model) et la transformation de ces modèles en modèles spécifiques aux plates-formes (PSM ou Platform Specific Model). L'idée est d'automatiser la projection d'un PIM vers un PSM, pour qu'en cas de changement d'architecture technique, il suffise de régénérer un autre PSM à partir du même PIM. Cette séparation entre les spécifications fonctionnelles et techniques permet d'assurer la réutilisation des PIM, leur portabilité, et aussi l'interopérabilité.

MDA est une approche basée sur les modèles et un ensemble de standards de l'OMG (UML, MOF, CWM, XML et IDL). L'Ingénierie Dirigée par les Modèles (IDM), appelée en anglais MDE (Model-Driven Engineering), est une démarche plus générale que MDA. Ainsi, MDA est donc un cas particulier de MDE (Favre, 2004). MDE considère l'existence de modèles dans le sens large (les méta-méta-modèles et les méta-modèles sont aussi des modèles) sur lesquels des opérations spécifiques peuvent être réalisées (Lopes, 2005). De plus, MDE est une approche ouverte qui prend en charge plusieurs autres espaces technologiques afin de les harmoniser (Favre 2004, Kurtev et al. 2002).

MDE peut être considéré comme une famille d'approches qui se développent à la fois dans les laboratoires de recherche et chez les industriels impliqués dans les grands projets de développement logiciels. Deux de ces industriels (IBM et Microsoft) ont récemment défini leur stratégie MDE et le moins que l'on puisse dire c'est qu'elles semblent converger (Bézivin et al., 2005).

Les techniques de MDE visent, en manipulant des modèles sous forme d'instances de méta-modèles, d'une part de générer l'essentiel du code des applications, et, d'autre part, de rendre les générateurs génériques par rapport à des familles de modèles. Actuellement, il y a plusieurs approches basées sur les principes de MDE, les plus connues sont "Software Factories" de Microsoft (GreenField et al., 2004), l'approche d'IBM (intégrée dans l'outillage (Eclipse Modeling Framework)) (Booch et al., 2004) et MDA de l'OMG (OMG, 2003d). L'approche Microsoft est fondée essentiellement sur des langages de domaines (Domaine Specific Languages ou DSL) (Cook, 2004) de petite taille, facilement manipulables, transformables, combinables, etc. Ces DSL représentent la base de l'automatisation de MDE chez Microsoft. Les DSL sont aussi utilisés dans l'approche IBM. En effet, les trois axes de l'ingénierie dirigée par les modèles d'après IBM (Booch et al., 2004) sont (1) les standards ouverts (UML, XML, etc.), (2) l'automatisation (outils pour le traitement automatique des modèles : des DSL adaptés à des corporations particulières ou à des besoins particuliers), (3) et la représentation directe (mise à disposition de métiers particuliers ou de tâches spécifiques de langages précis et outillés) (Bézivin et al., 2005).

Le MDE est une approche prometteuse qui devrait permettre d'améliorer le développement de logiciel grâce à 4 points forts qui sont (Jézéquel et al., 2005) :

- la possibilité de capitaliser sur un processus de développement en automatisant les raffinements.
- la capitalisation du savoir-faire de conception grâce à la définition et l'implantation de transformations de modèles.
- la réutilisation à un niveau abstrait grâce à la formalisation des artefacts produits au cours du développement en tant que modèles.
- la possibilité de choisir la solution technologique la plus adaptée à chaque activité du processus grâce aux ponts entre domaines technologiques.

Comme VUML est une extension d'UML, nous allons, dans la suite, nous concentrer sur l'approche MDA.

Afin de profiter des avantages apportés par l'approche MDA, nous avons décidé d'associer à VUML une démarche dirigée par les modèles. Ceci va permettre la pérennité de la logique métier grâce à l'élaboration de modèles, la productivité de cette logique métier grâce à l'automatisation des transformations de modèles, et la prise en compte des plates-formes d'exécution grâce à l'intégration de celles-ci dans les transformations de modèles.

Dans ce chapitre, nous présentons tout d'abord les grandes lignes de l'approche MDA. Le lecteur trouvera une description complète de cette approche dans (OMG 2003d, Blanc 2005). Ensuite, nous décrivons globalement la démarche associée à VUML dont un noyau a été présenté dans (Nassar, 2004). Ce chapitre a pour objectif de situer cette démarche dans le contexte MDA. Nous nous appuyons sur l'exemple du SED (Système d'Enseignement à Distance) pour illustrer notre approche.

V.2. L'approche MDA

V.2.1. Introduction

L'objectif majeur de MDA est l'élaboration de modèles pérennes, indépendants des plates-formes d'exécution, afin de permettre la génération automatique du code des applications et d'améliorer la productivité (pour qu'un modèle soit "productif" il doit pouvoir être interprétable et manipulable par une machine (Bézivin et al., 2005)). A cette fin, le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, MDA préconise l'élaboration de modèles d'exigences CIM (Computation Independent Model), d'analyse et de conception PIM (Plateform Independent Model) et de code PSM (Plateform Specific Model).

MDA vise à résoudre les problèmes d'interopérabilité et de portabilité dès le niveau modélisation. En effet, la solution apportée par le standard CORBA pour faciliter la construction et la maintenance des applications réparties n'a pas été un franc succès. MDA permet ainsi de concevoir des applications portables au niveau des langages de programmation, des systèmes d'exploitation mais aussi des middlewares. Ceci permet de capitaliser le travail effectué lors de l'analyse/conception.

La figure 94 représente le logo du MDA qui illustre les différentes couches de spécifications. Dans le noyau se trouvent les techniques UML, MOF, CWM. La première couche représente les plates-formes supportées. La deuxième couche concerne les services systèmes et enfin l'extérieur montre les domaines pour lesquels des composants métiers doivent être définis (Domain Facilities).

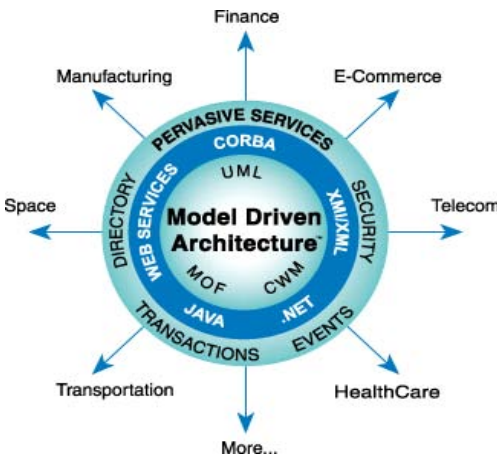

Figure 94 – *Le Model Driven Architecture*

V.2.2. Mise en œuvre du MDA

La démarche MDA supporte tout le cycle de développement. Elle peut se diviser en cinq étapes (Blanc 2005, Projet Accord 2002, OMG 2003d). La figure 95 illustre ces cinq étapes :

1- Elaboration d'un ou plusieurs modèles d'exigences (CIM).

2- Elaboration des modèles indépendants de toute plate-forme (PIM). Ces modèles sont aussi appelés « modèles d'analyse et de conception abstraite ». Ils doivent en théorie partiellement être générés à partir des modèles CIM afin de garantir l'établissement des liens de traçabilité.

3- Enrichissement du modèle PIM par étapes successives.

4- Choix des plates-forme de mise en œuvre et génération des modèles spécifiques correspondants (PSM). Ces modèles sont obtenus par une transformation des PIM en y ajoutant les informations techniques relatives aux plates-forme choisies.

5- Raffinement des PSM jusqu'à l'obtention d'une implémentation exécutable.

Les étapes 3 et 5 peuvent se répéter un nombre de fois indéterminé. La génération de code à partir des modèles PSM n'est pas réellement considérée par MDA. Celle-ci s'apparente plutôt à une traduction des PSM dans un formalisme textuel. D'autre part, MDA peut procurer de nombreux avantages pour la rétroconception d'applications existantes. C'est pourquoi les transformations inverses (PSM vers PIM et PIM vers CIM) sont aussi utiles. Ces transformations n'en sont toutefois encore qu'au stade de la recherche (Blanc, 2005).

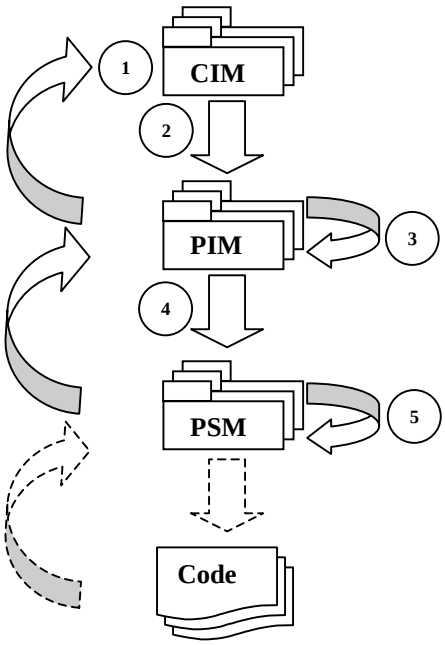

Figure 95 – *Les étapes d'une démarche MDA*

V.2.2.1. Computation Independent Model (CIM)

L'objectif de MDA est de généraliser l'utilisation des modèles dans toutes les phases du cycle de développement. Bien que très en amont, la spécification des exigences donne lieu aussi à des modèles, et la création d'un modèle d'exigences est d'une importance capitale. Ce modèle permet d'exprimer les liens de traçabilité avec les modèles qui seront élaborés dans les autres phases du cycle de développement. Un modèle d'exigences ne contient pas d'information sur l'implémentation de

l'application ni sur les traitements. Ceci justifie le fait que ces modèles sont appelés des modèles indépendants de la programmation (CIM : Computation Independent Model).

Dans UML, les modèles d'exigences sont représentés par les diagrammes de cas d'utilisation. Ces derniers décrivent de façon abstraite les différentes fonctionnalités du système ainsi que les acteurs qui interagissent avec ce système. Cependant, MDA ne fait aucune préconisation quant à l'utilisation d'UML ou non pour exprimer les CIM. En effet, il existe d'autres approches d'expression des exigences, telles que celles supportées par DOORS ou Rational Requisite Pro (Blanc, 2005).

V.2.2.2. Platform Independent Model (PIM)

La deuxième étape de la démarche MDA consiste à réaliser des modèles indépendants de toute plate-forme (PIM). MDA considère que les modèles d'analyse et de conception doivent être indépendants des plates-formes d'implémentation. En effet, en intégrant les détails d'implémentation tardivement dans le cycle de développement, il est possible de maximiser la séparation des préoccupations entre la logique de l'application et les techniques d'implémentation. Dans cette deuxième étape de la démarche MDA, il est possible d'élaborer plusieurs modèles PIM. Chacun de ces modèles appartient à un niveau de PIM mais tous sont indépendants de n'importe quelle plate-forme. Le PIM de base représente uniquement les capacités fonctionnelles métiers et le comportement de l'application. Ce modèle doit être clair afin de permettre à des experts du domaine de le comprendre plus aisément qu'un modèle d'implémentation. Ils peuvent ainsi vérifier plus facilement que le PIM est complet et correct. Les PIM suivants ajoutent des aspects technologiques et architecturaux mais toujours sans détails spécifiques à une plate-forme. Par exemple, ces modèles peuvent intégrer des informations sur la persistance, les transactions, la sécurité, etc. Précisons que MDA ne donne aucune indication quant au nombre de modèles à élaborer ni quant à la méthode à utiliser pour élaborer les PIM.

UML est préconisé par MDA comme étant le langage à utiliser pour réaliser des PIM. Toutefois, MDA ne fait que préconiser l'utilisation de ce langage de modélisation et il n'exclut pas que d'autres langages puissent être utilisés. L'essentiel pour un PIM est d'être pérenne et de faire le lien entre le modèle d'exigences et le code de l'application. Les modèles doivent aussi être productifs car ils constituent le socle du processus de génération de code.

V.2.2.3. Platform Specific Model (PSM)

Une fois les modèles PIM réalisés (suffisamment détaillés), il faut prendre en compte les plates-formes cibles. Ce travail consiste à projeter les PIM vers des modèles spécifiques à des plates-formes (PSM). En fait, les PSM combinent les spécifications intégrées dans les PIM et les détails techniques concernant une plate-forme donnée.

Les PSM servent principalement à faciliter la génération de code à partir d'un modèle d'analyse et de conception. Ces modèles contiennent toutes les informations nécessaires à l'exploitation d'une plate-forme d'exécution. Ils sont donc essentiellement productifs mais pas forcément pérennes. Comme pour les PIM, il existe plusieurs niveaux de PSM. Le premier est obtenu directement à partir du modèle PIM, les autres sont obtenus par transformations successives jusqu'à l'obtention d'un système exécutable.

MDA conseille l'utilisation des profils UML pour élaborer les PSM. En effet, les profils UML permettent d'adapter UML à un domaine particulier. Par exemple, le profil UML pour EJB, le profil

UML pour CORBA, etc. L'utilisation de ces profils a le mérite de faciliter les transformations PIM vers PSM car PIM et PSM sont tous deux des modèles UML.

Une deuxième approche pour élaborer les PSM consiste en la définition de méta-modèles propres aux plates-formes. Cette approche offre une plus grande liberté dans l'expression des plates-fromes ; cependant, elle présente l'inconvénient de ne pas faciliter les transformations PIM vers PSM (Blanc, 2005).

V.2.3. Les bases techniques

Au cœur de l'approche MDA, se trouvent plusieurs standards de l'OMG : le Meta Object Facility (MOF), Unified Modeling Language (UML), le Common Warehouse Metamodel (CWM), et XML Metadata Interchange (XMI). Le standard MOF (OMG, 2002a) apporte le support de définition des formalismes de modélisation sous la forme de méta-modèles. Le langage UML (OMG, 2003a) permet de modéliser une application indépendamment de toute démarche et de toute plate-forme. C'est pour cette raison que MDA préconise UML pour décrire les PIM. Le CWM (OMG, 2003e) est le standard de l'OMG pour les techniques liées aux entrepôts de données. Le standard XMI (OMG, 2002b) quant à lui permet de représenter les modèles sous forme de documents XML. En fait, MDA définit deux façons différentes pour représenter les modèles : soit sous forme de documents textuels soit sous forme d'objets de programmation. La représentation textuelle est adaptée pour le stockage des modèles sur des mémoires de masse ou l'échange des modèles entre applications. La représentation objet est plus adéquate pour les transformations, exécution, et validation des modèles.

En plus de XMI, MDA offre deux autres standards pour la représentation des modèles : JMI (Java Metadata Interface), et EMF (Eclipse Modeling Framework). Ces deux standards définissent la façon de représenter les modèles sous forme d'objets Java. Le principe de fonctionnement de XMI, JMI et EMF est de générer automatiquement la structure des formats de représentation des modèles à partir de leur méta-modèle. Ces standards tirent partie de l'analogie qui existe entre la relation entre modèle et méta-modèle et la relation qui existe entre un document XML et sa DTD ou entre les objets et leur classe.

D'autre part, et afin d'avoir un moyen pour modéliser les corps des opérations UML, deux standards sont utilisés à savoir : OCL (Object Constraint Language) et AS (Action Semantic). L'approche OCL consiste à définir des contraintes sur les opérations afin de préciser ce que doit être la sortie d'une opération en fonction de son entrée. L'approche AS consiste à définir une suite d'actions modifiant l'état d'un modèle. Cependant, il n'y a pas à ce jour de syntaxe concrète standardisée pour AS.

V.2.4. Transformation de modèles

La productivité des modèles est l'un des apports essentiels de l'approche MDA. Rendre les modèles productifs consiste à leur appliquer des transformations de modèles pour obtenir des résultats utiles au développement. Ces résultats sont, en général, des modèles plus détaillés et proches de la solution technique. Il faut noter que ce sont ces transformations qui portent l'intelligence du processus méthodologique de construction d'applications.

V.2.4.1. Types de transformations de modèles

La figure 95 ci-dessus illustre les principales transformations de modèles MDA. Les transformations préconisées par MDA sont essentiellement les transformations CIM vers PIM et PIM vers PSM. Les transformations inverses ne font pas partie de l'objectif actuel de MDA mais elles seront fortement mises à contribution dans l'élaboration du futur standard ADM (Architecture Driven Modernization) de l'OMG. Ce standard vise à définir l'approche inverse de MDA ce qui va permettre d'élaborer des modèles à partir des applications existantes (Blanc, 2005).

Transformations CIM vers PIM : Ces transformations visent à construire des modèles PIM partiels à partir des modèles CIM. Elles permettent de garantir la traçabilité entre les modèles et les besoins exprimés dans les CIM.

Transformations PIM vers PIM : Ce genre de transformation permet de raffiner les PIM afin de les enrichir par des informations plus précises.

Transformations PIM vers PSM : Ce type de transformation est utilisé quand les PIM sont suffisamment raffinés pour être projetés vers une plate-forme d'exécution donnée. Ces transformations permettent de construire une bonne partie des modèles PSM à partir des modèles PIM. Elles sont les plus importantes de l'approche MDA car elles garantissent la pérennité des modèles aussi bien que leur productivité et leur lien avec les plates-formes d'exécution.

Transformations PSM vers code : Ces transformations s'appliquent sur un PSM et permettent de générer le code associé à l'application.

V.2.4.2. Métamodèles et règles de correspondance

Une transformation de modèles MDA est considérée comme une fonction qui prend en entrée un ensemble de modèles et qui fournit en sortie un ensemble de modèles. Ces modèles d'entrée et de sortie doivent être conformes à des métamodèles. L'exécution d'une transformation de modèle se fait au niveau des modèles, mais elle se spécifie au niveau des métamodèles. En effet, une transformation exprime des correspondances structurelles entre les modèles sources et cibles. Ces correspondances s'appuient sur les métamodèles de ces modèles source et cible.

La figure 96 ci-après illustre un exemple de métamodèles utilisés dans une transformation de modèles. Cette transformation vise à transformer des modèles UML vers des schémas de bases de données relationnelles (BD). Cette transformation est basée sur une règle de correspondance structurelle entre les métamodèles UML et BD. Au niveau modèles, la règle de correspondance est la suivante : à toute classe UML correspond une table d'un schéma d'une base de données. Cette même règle au niveau métamodèles devient : à tout élément du modèle source instance de la métaclasse *Class* du métamodèle UML correspond un élément du modèle cible instance de la métaclasse *Table* du métamodèle BD (supposé défini).

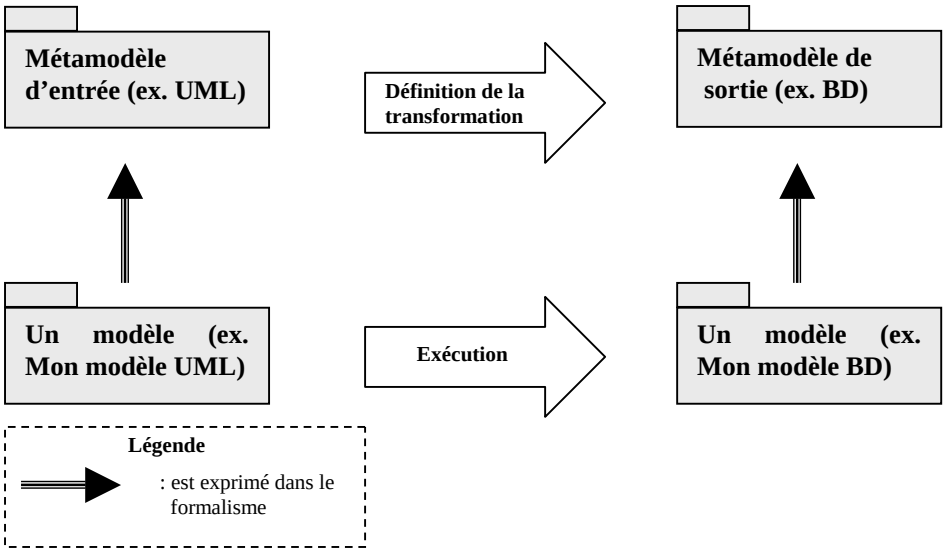

Figure 96 – *Métamodèles et transformation de modèles (Blanc, 2005)*

V.2.4.3. Spécification des règles de transformation

Dans (OMG, 2003d), les auteurs présentent une classification des approches de transformations de modèles : *marking*, transformation de modèles, application de patrons et fusion de modèles.

Actuellement, la création d'un langage de transformation normalisé est le centre des réflexions autour de MDA. Dans ce contexte, en 2002, l'OMG a lancé un appel à travers son RFP QVT (OMG, 2002d) pour que des propositions d'un langage de transformations soient faites. Plusieurs langages ont été proposés (DSTC 2004, IOS GMBH 2004, QVT-MERGE GROUP 2004) mais ils sont encore en phase d'analyse.

Il existe trois approches de transformations de modèles : approche par programmation, approche par template, et approche par modélisation. La différence entre ces trois approches réside dans la façon dont sont spécifiées les règles de transformations.

Approche par programmation : cette approche utilise les langages de programmation objet pour implémenter les transformations. Ainsi les transformations sont des programmes qui manipulent des modèles. C'est l'approche la plus utilisée car elle est très puissante et fortement outillée (IBM Rational Software Modeler, Softeam MDA Modeler).

Approche par template : elle consiste en l'élaboration de canevas des modèles cibles en y déclarant des paramètres. Ces paramètres seront substitués par les informations contenues dans les modèles sources. Les canevas sont appelés « modèles templates ». Cette approche nécessite des langages particuliers qui permettent la définition des modèles templates.

Les outils qui supportent l'approche par template proposent deux types de langages :

- Les langages graphiques spécifiques d'un méta-modèle : ces langages sont, en général, faciles à utiliser mais peu expressifs car ils dépendent d'un seul méta-modèle. Par conséquent, ces langages ne peuvent transformer qu'un seul type de modèle. UML est un type de langage graphique spécifique d'un méta-modèle. En effet, UML permet de définir des modèles paramétrés. Les paramètres de ces modèles seront substitués par des informations contenues dans des modèles sources.

- Les langages textuels indépendants des métamodèles : ils sont très expressifs mais plus difficiles à utiliser. Ces langages ont un caractère générique et permettent de transformer n'importe quel modèle. Les langages fondés sur XMI (XML Metadata Interchange) et sur les langages classiques sont des exemples de langages indépendants des métamodèles.

Bien que l'approche par template soit très outillée, il est difficile d'identifier avec certitude l'apport de cette approche par rapport à l'approche par programmation.

Approche par modélisation : cette approche vise à appliquer l'ingénierie des modèles aux transformations de modèles, afin de les rendre pérennes, productifs et indépendants des plates-formes d'exécution. Le standard MOF2.0 QVT (Query, View, Transformation) sera bientôt défini par l'OMG. Ce standard va offrir un métamodèle permettant de construire des modèles de transformations de modèles. Une vision très simplifiée de ce métamodèle est présentée sur la figure 97. Ce métamodèle est constitué d'un certain nombre de métclasses reliées par des méta-associations. La métaclasse *Module* représente une transformation de modèles. Elle est reliée par deux méta-associations (sources et cibles) à la métaclasse *Package* qui représente un métamodèle. Un *Module* est composé de trois méta-classes : *Query*, *Relation*, et *Mapping*. La métaclasse *Query* représente les requêtes effectuées dans une transformation de modèles. Ces requêtes sont des expressions OCL (sans effet de bord) permettant de naviguer dans les modèles pour obtenir certaines informations. La métaclasse *Relation* représente des règles de correspondance entre les parties structurelles des métamodèles source et cible. Ces règles concernent des sous-ensembles d'un métamodèle (représentés par des métaclasses *Domain*). Elles expriment seulement des correspondances structurelles entre les métamodèles sans définir la façon dont sont réalisées ces correspondances. La métaclasse *Mapping* quant à elle représente des règles de construction. Cette métaclasse est reliée à la métaclasse *MatchingExpression*, qui représente le concept d'action de construction.

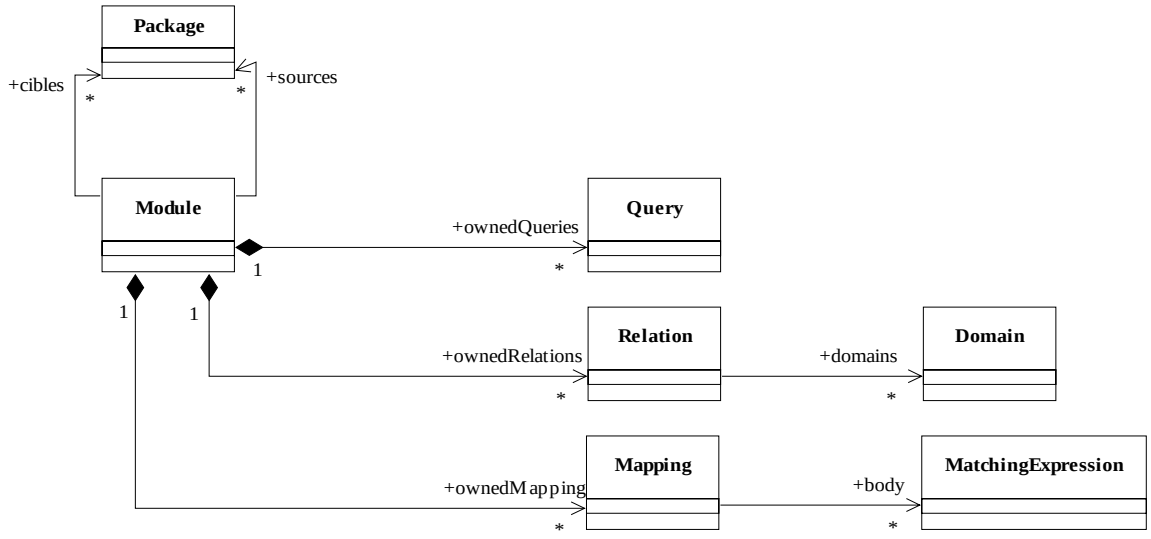

Figure 97 – *Vision simplifiée du métamodèle MOF2.0 QVT (Blanc, 2005)*

La figure 98 présente à titre d'exemple les relations existantes entre le métamodèle MOF2.0 QVT, un modèle de transformation (UML2BD), son métamodèle d'entrée (UML) et de sortie (BD) et une exécution du modèle de transformation.

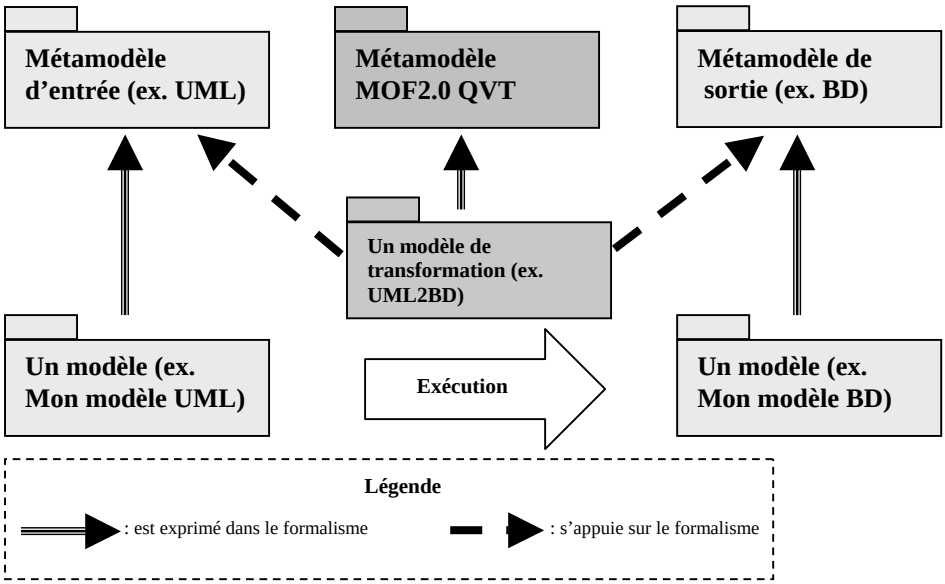

Figure 98 – *Exemple de transformation de modèles avec QVT (Blanc, 2005)*

Plusieurs langages de transformations de modèles ont été proposés, parmi lesquels nous pouvons citer : QVT Merge (Grønmo et al., 2005), ATL (Atlas Transformation Language) (Bézivin et al., 2003), YATL (Yet Another Tranformation Language) (Patrascoiu, 2004), BOTL (Basic Object-oriented Transformation Language) (Marschall et al., 2003), etc. Ces langages sont basés sur différents paradigmes et un langage peut être adapté qu'un autre à un contexte spécifique (Czarnecki et al., 2003).

Pour illustrer l'utilisation d'un langage de transformation, nous avons choisi le langage ATL[6] qui est outillé et largement expérimenté. C'est un langage hybride (déclaratif et impératif). L'approche déclarative d'ATL est basée sur OCL. L'approche impérative en ATL contient des instructions qui explicitent les étapes d'exécution dans des procédures (*helpers*). La figure 99 présente la syntaxe abstraite de règles de transformation en ATL. Cette figure montre qu'une règle est identifiée par un nom et peut contenir les éléments *ActionBlock* et *OutPattern*. Une *MatchedRule* spécialise une *Rule* et contient un *InPattern*. Une *CalledRule* spécialise une *Rule* ou une *Operation* en OCL.

La version actuelle d'ATL est basé sur MDR (MetaData Repository) (NetBeans.ORG, 2003), EMF (Eclipse Modeling Framework), Java et Eclipse (ATL, 2004).

Parmi les travaux récents basés sur ATL, nous citons ceux de Lopes (Lopes, 2005) qui propose une méthodologie pour le développement d'applications selon l'approche MDA. Cette méthodologie est basée sur une architecture type pour la transformation de modèles qui distingue explicitement la spécification de correspondances et la définition de transformations. La correspondance décrit quels sont les éléments équivalents ou similaires entre deux métamodèles. La transformation a pour but de transformer un élément de modèle source en un élément de modèle cible conformément à la correspondance (mapping) associée, en fonction d'une plate-forme cible.

[6] ATL a été conçu, développé et implémenté par J. Bézivin et F. Jouault

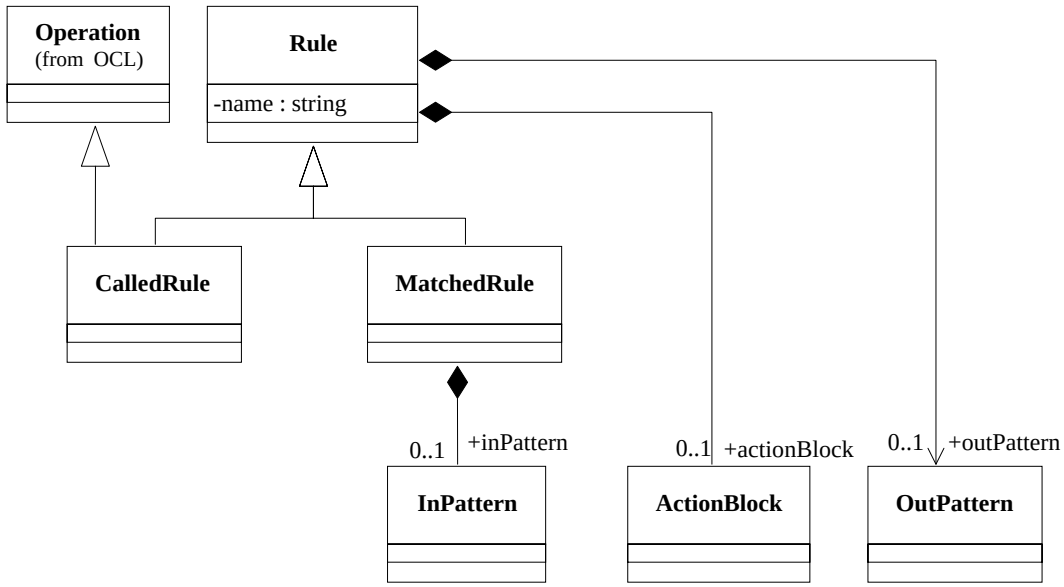

Figure 99 – *Règles en ATL (syntaxe abstraite) (Bézivin et al., 2003)*

Pour spécifier les correspondances entre les métamodèles source et cible, Lopes propose une notation graphique (qui réutilise la notation UML) avec l'utilisation d'OCL (Object Constraint Language) pour exécuter des requêtes sur les métamodèles. La définition de transformations est réalisée automatiquement en ATL à partir de la spécification des correspondances. Dans cette perspective, la spécification de correspondance peut être vue comme un PIM et la définition de transformation comme un PSM.

V.2.5. Synthèse

Dans cette section, nous avons présenté les grandes lignes de l'approche MDA. Cette approche utilise fortement les modèles. MDA a pour objectif la séparation des préoccupations entre le métier des applications et la technique des intergiciels. Les avantages attendus de MDA sont (Blanc, 2005) :

- La pérennité des savoir-faire : ce qui va permettre aux entreprises de capitaliser les connaissances sur leur métier sans avoir à se soucier de la technique.
- Les gains de productivité : ce qui va permettre aux entreprises de réduire les coûts de mise en œuvre des applications.
- La prise en compte des plates-formes d'exécution : ce qui va permettre aux entreprises de bénéficier des avantages des plates-formes.

V.3. Vers une démarche VUML dirigée par les modèles

Le processus de développement avec VUML n'étant pas l'objectif principal de cette thèse, nous ne présentons qu'une version simplifiée de la démarche associée à VUML. Les phases principales de cette démarche sont inspirées de la méthode VBOOM (Kriouile, 1995). La démarche associée à VUML s'intègre aussi dans l'approche MDA dont les étapes sont présentées dans la section V.2.2. Dans cette section, nous allons situer les différents modèles VUML dans les niveaux proposés par MDA (CIM,

PIM, PSM). Concernant les transformations de modèles, nous nous limitons à donner les principes des différentes transformations de modèles VUML. Un travail de thèse a été lancé récemment sur l'affinage de ces transformations au sein de la démarche VUML.

La modélisation VUML s'effectue en 4 phases (cf. figure 100) : phase centralisée de modélisation des exigences, phase décentralisée de modélisation du système suivant chaque point de vue, phase centralisée de fusion et de modélisation produisant le diagramme de classes VUML, phase de prise en compte des plates-formes d'exécution.

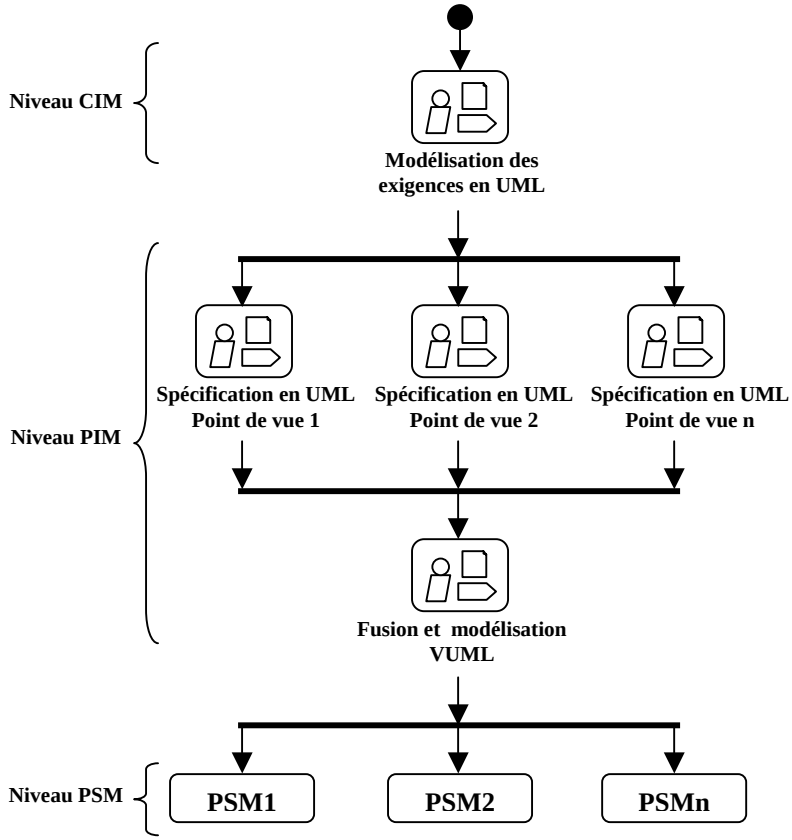

Figure 100 – *Démarche par point de vue dirigée par les modèles associée à VUML*

V.3.1. Modélisation des exigences (Niveau CIM)

La phase de modélisation des exigences est une phase centralisée qui fait partie du niveau CIM (Computation Independent Model) de l'approche MDA. Cette phase est principalement réalisée par les cas d'utilisation. Elle permet d'identifier les différents acteurs (points de vue) qui interagissent avec l'application ainsi que les besoins de chaque acteur. De plus, des liens de traçabilité peuvent être créés depuis les exigences vers le code de l'application.

V.3.2. Modélisation UML par point de vue (Niveau PIM)

La deuxième phase de la démarche associée à VUML est l'élaboration des modèles (PIM) par points de vue — en utilisant les différents diagrammes UML. Ces PIM par point de vue devraient en théorie être générés à partir des modèles CIM élaborés dans la phase de modélisation des exigences.

Cette génération devrait permettre de garantir l'établissement des liens de traçabilité, en utilisant des transformations de modèles (CIM->PIM) qui tiennent compte du point de vue concerné (figure 101).

Dans la réalité, cette génération n'est souvent que très partielle, et nous ne la prenons pas en considération pour l'instant.

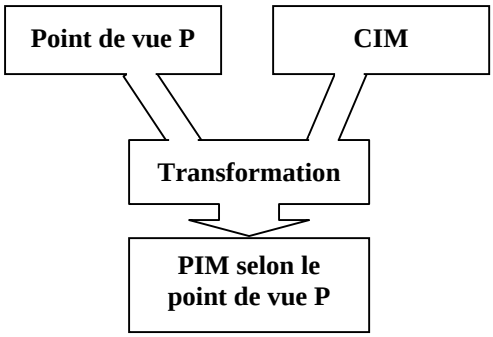

Figure 101 – *Transformation CIM->PIM par point de vue*

Cette phase consiste à faire une spécification en UML par point de vue c'est-à-dire par acteur. Pour chaque acteur, nous élaborons les diagrammes de séquence de chaque cas d'utilisation auquel il participe. A partir de ces diagrammes de séquences nous identifions les classes et les méthodes du diagramme de classes associé à cet acteur. Un processus itératif de type RUP (Rational Unified Process) est appliqué ainsi pour chaque acteur. Le résultat de cette phase est un diagramme de classes par point de vue qui montre clairement les informations pertinentes auxquelles peut accéder l'acteur associé. Les différents PIM par points de vue élaborés dans la phase de modélisation par points de vue doivent être enrichis par étapes successives (cf. section V.2.2).

V.3.3. Fusion et modélisation VUML (PIM→ PIM)

Une fois les PIM par point de vue suffisamment détaillés, la phase de fusion de ces modèles peut commencer. Cette fusion peut être considérée comme une transformation de modèles (PIM-> PIM). En effet, une telle transformation prend en entrée des PIM par points de vue exprimés en UML et génère en sortie un PIM exprimé en VUML (cf. figure 102). Dans cette section nous ne donnons que les grandes lignes de cette fusion. Une étude approfondie est en cours pour mettre en oeuvre une telle transformation. La première étape de la fusion est la comparaison des diagrammes de classes par points de vue qui peut révéler des ambiguïtés ou incohérences sur les classes (nommage, attributs, méthodes). Les diagrammes de classes élaborés précédemment peuvent être fusionnés ensuite en un seul diagramme. Si l'on faisait cette fusion de manière classique en UML — ce qui serait possible — cela reviendrait à perdre les informations liées aux caractéristiques de chaque point de vue : en faisant apparaître simultanément toutes les classes et tous les attributs, on ne spécifierait plus quel acteur peut accéder à ces informations. En revanche, avec la fusion VUML, il est possible de modéliser sur un seul diagramme de classes la totalité des informations présentes dans chaque point de vue, sans pour autant perdre les spécificités de chaque acteur. Une fois le modèle VUML élaboré, il faut décrire les éventuelles dépendances entre les vues des différentes classes multivues. Rappelons que dans VUML, ces dépendances sont modélisées par des relations de dépendance stéréotypées par « viewDependency », annotées par des contraintes (exprimées soit en OCL soit d'une manière textuelle). Ces relations seront exploitées lors de la phase d'implémentation pour gérer la cohérence entre les vues. En poursuivant ce travail de fusion, on obtient un diagramme de classes VUML où l'on

peut, selon le nombre de classes et de vues, présenter les classes multivues soit sous forme éclatée en faisant apparaître chaque vue, soit sous forme "iconifiée" en spécifiant le stéréotype « multiViewsClass ». Cette spécification permet par la suite de développer des composants comme cela a été présenté en section II.4 et de les réaliser à l'aide du patron d'implantation décrit dans la section II.5.5.

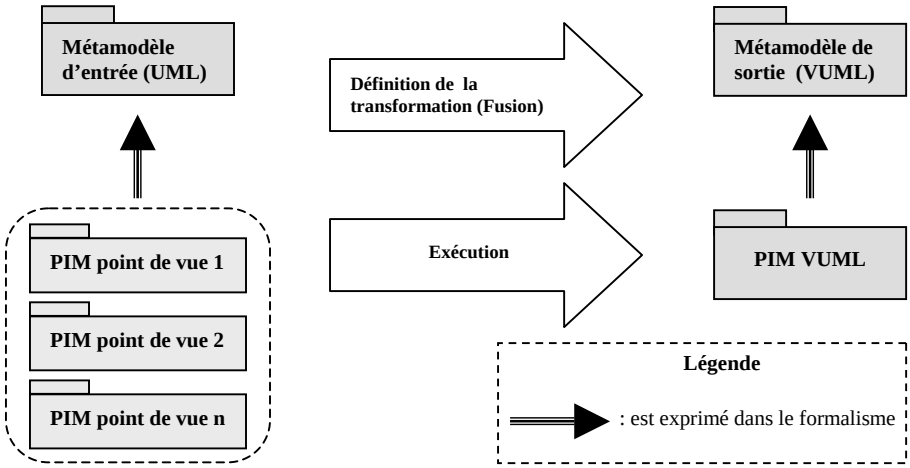

Figure 102 – *Transformation de modèles PIM par point de vue (Fusion – scénario 1)*

Nous pouvons aussi imaginer d'autres scénarios pour réaliser la fusion des différents PIM par point de vue. A titre d'exemple, la figure 103 illustre la fusion de 4 PIM par point de vues en utilisant un processus de fusion incrémental. A chaque étape, deux PIM sont fusionnés pour donner lieu à un PIM VUML. Ce dernier est ensuite fusionné avec un autre PIM par point de vue pour donner lieu à un autre PIM VUML et ainsi de suite jusqu'à l'élaboration du PIM VUML global.

V.3.4. Ajout d'un point de vue

VUML offre la possibilité d'ajouter à tout moment dans le modèle un nouveau point de vue qui donnera lieu à une nouvelle vue pour une ou plusieurs classes multivues. Cette évolution du modèle ne doit pas remettre en cause le modèle déjà existant. Au sens de MDA, l'ajout d'un point de vue est comparable à une transformation de modèles qui permet (cf. figure 104) de fusionner un PIM VUML (modèle VUML avant l'ajout du point de vue) et un PIM par point de vue (correspondant au point de vue à ajouter). Le résultat est un autre PIM VUML. Cette transformation peut être réalisée en exploitant le mécanisme de spécialisation des classes multivues (cf. section II.3.4) qui offre la possibilité d'ajouter des points de vues à une classe multivues.

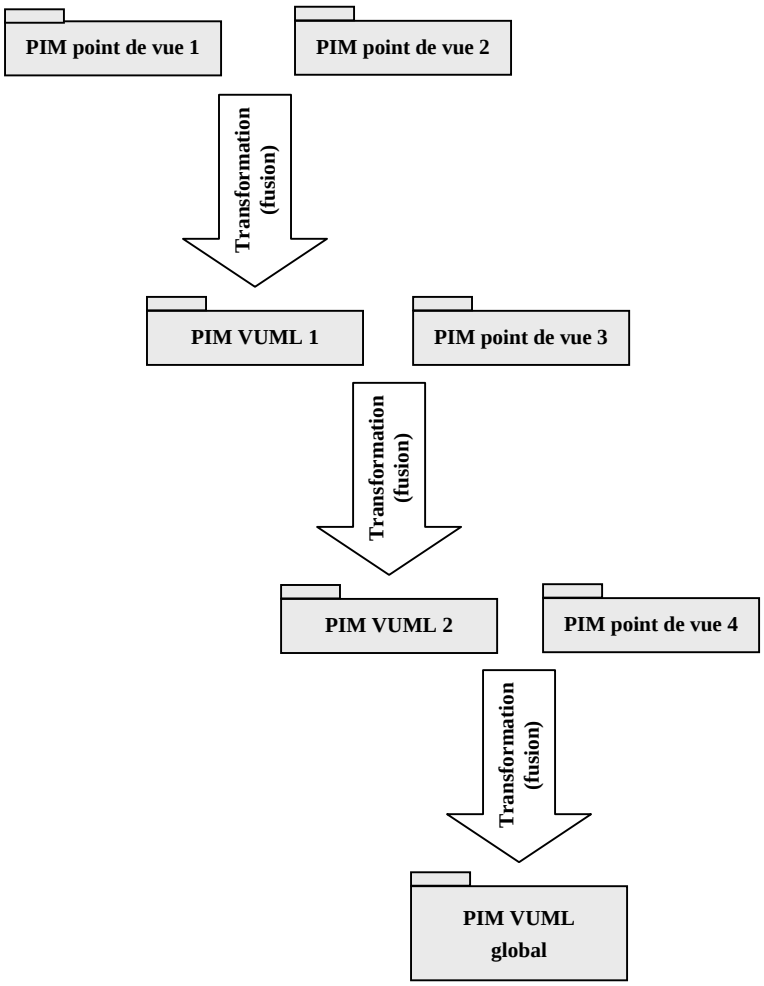

Figure 103 – *Transformation de modèles PIM par point de vue (Fusion – scénario 2)*

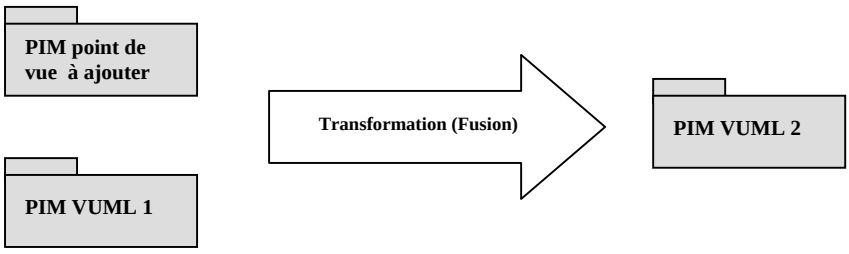

Figure 104 – *Transformation de modèles (Ajout d'un point de vue)*

V.3.5. Prise en compte des plates-formes d'exécution (PIM → PSM)

La prise en compte des plates-formes d'exécution dans le cycle de vie des applications a pour objectif de gérer la dépendance des applications vis-à-vis de leur plate-forme d'exécution. Dans MDA, ces dépendances sont gérées comme une transformation de modèles. Nous préconisons l'utilisation de patrons relatifs aux plates-formes pour réaliser des transformations qui transforment des PIM VUML

vers des PSM (cf. figure 105). Actuellement, la démarche VUML offre un patron qui permet de transformer un modèle PIM VUML en un modèle PSM spécifique à des plates-formes orientées objet. Ce patron a été présenté dans la section II.5.5. Le modèle PSM généré peut être facilement utilisé pour générer du code dans des langages orientée objet (Java, C++, Eiffel, …), soit par un processus de transformation MDA, soit par programmation.

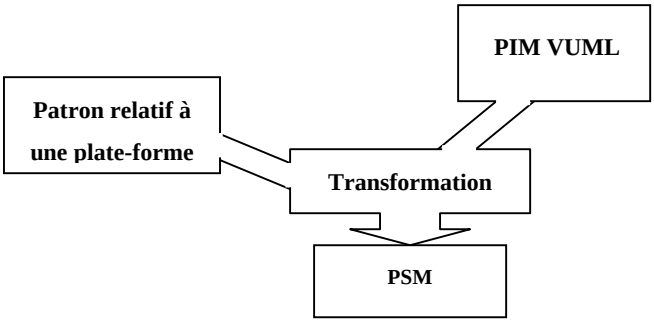

Figure 105 – *Transformation PIM VUML -> PSM en utilisant des patrons relatifs aux plates-formes*

V.4. Application : Système d'Enseignement à Distance (SED)

Afin d'illustrer l'utilisation de la démarche, nous nous appuyons sur la modélisation de l'étude de cas SED (Système d'Enseignement à Distance) présentée dans le chapitre 4.

Modélisation des exigences

Un résumé de l'analyse des exigences du SED a déjà été présenté dans la section IV.3. Le digramme de cas d'utilisation élaboré (cf. figure 106) constitue le modèle CIM qui sera utilisé pour produire les différents modèles par point de vue PIM.

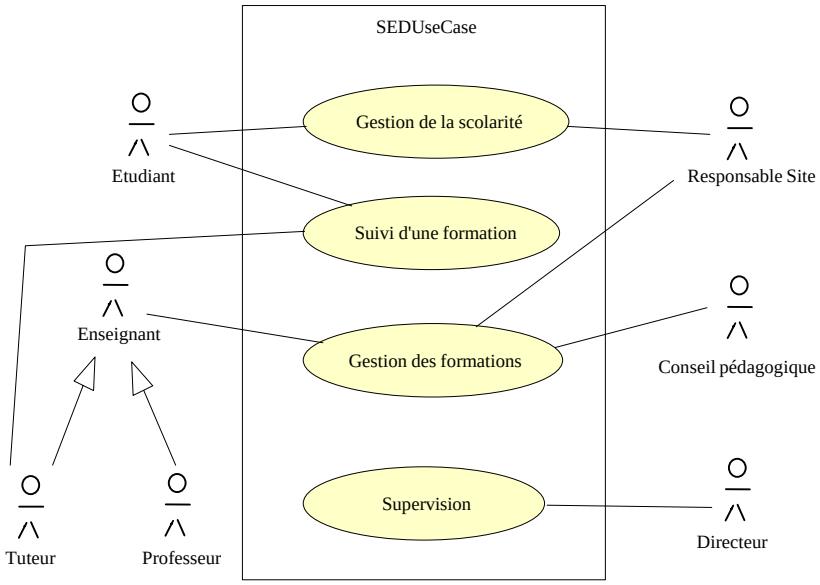

Figure 106 – *Cas d'utilisation du SED (Extrait)*

Spécification par point de vue

En théorie une partie des modèles par point de vue devrait être générée par des transformations de modèles (CIM->PIM). Cependant, dans notre cas, nous avons procédé manuellement à la réalisation de ces modèles PIM. Ainsi, pour chaque point de vue, il faut modéliser la dynamique du système en utilisant des diagrammes de séquence ou de collaboration. A titre d'exemple, la figure 107 présente le diagramme de séquence d'un scénario du cas d'utilisation *Gestion des formations*. Les objets identifiés ainsi que les méthodes associées sont reportés dans le diagramme de classes du point de vue *Enseignant* de la figure 108. De plus, ce diagramme fait apparaître des objets et donc d'autres classes identifiées lors des spécifications des autres cas d'utilisation (comme par exemple les classes *Question*, *Reponse* et *Etudiant*).

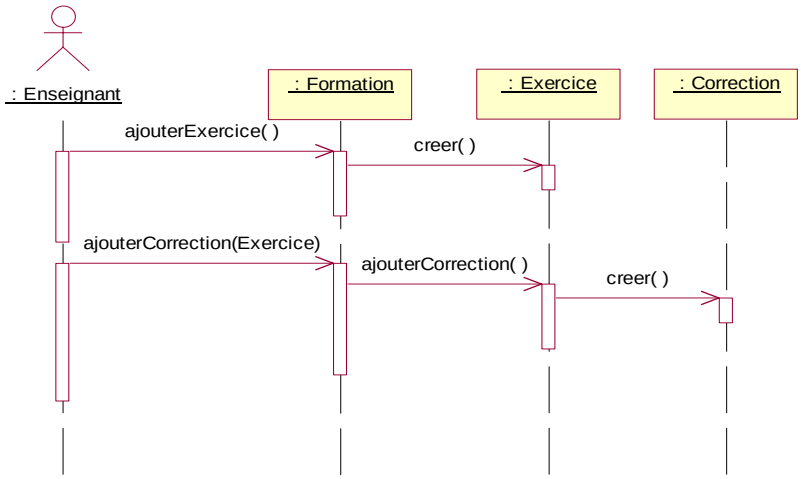

Figure 107 – *Diagramme de séquence d'un scénario du cas d'utilisation Gestion des formations*

Une démarche similaire effectuée pour chaque acteur et pour chaque cas d'utilisation est appliquée et donne lieu aux diagrammes de classes des figures 109 et 110. Afin de simplifier ces diagrammes, seuls quelques méthodes et attributs représentatifs sont présentés. On obtient ainsi un diagramme de classes par point de vue qui montre clairement les informations pertinentes auxquelles peut accéder l'acteur associé. Nous pouvons remarquer par exemple qu'un enseignant (cf. figure 108) doit avoir accès aux informations sur les étudiants inscrits à sa formation, et savoir quel est l'auteur de chaque question reçue. Ce n'est pas le cas de l'étudiant (cf. figure 109) qui peut simplement accéder à la documentation, aux exercices et à l'examen, et poser des questions. Il peut aussi demander de s'inscrire dans une formation. Le responsable de site (cf. figure 110) a besoin quant à lui, de connaître le responsable de la formation, la date de l'examen, et les étudiants inscrits.

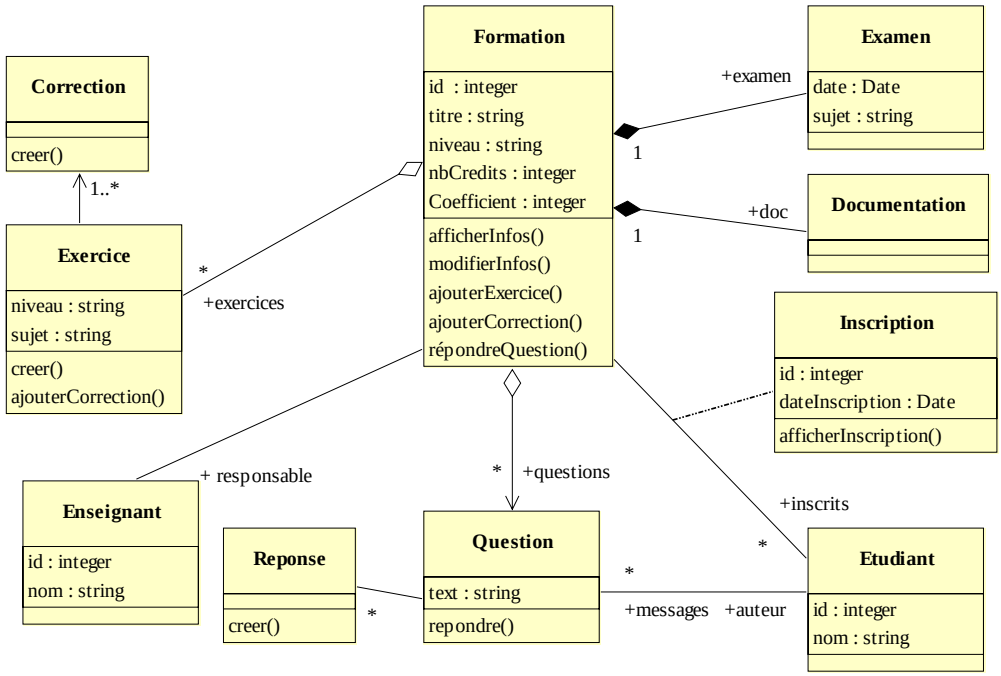

Figure 108 – *Diagramme de classes - Point de vue Enseignant*

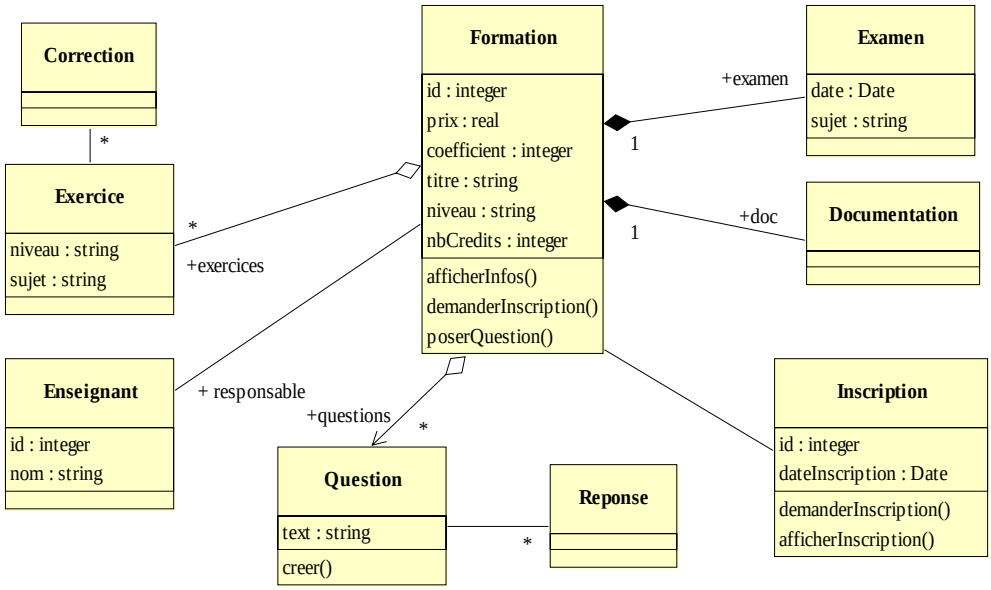

Figure 109 – *Diagramme de classes - Point de vue Etudiant*

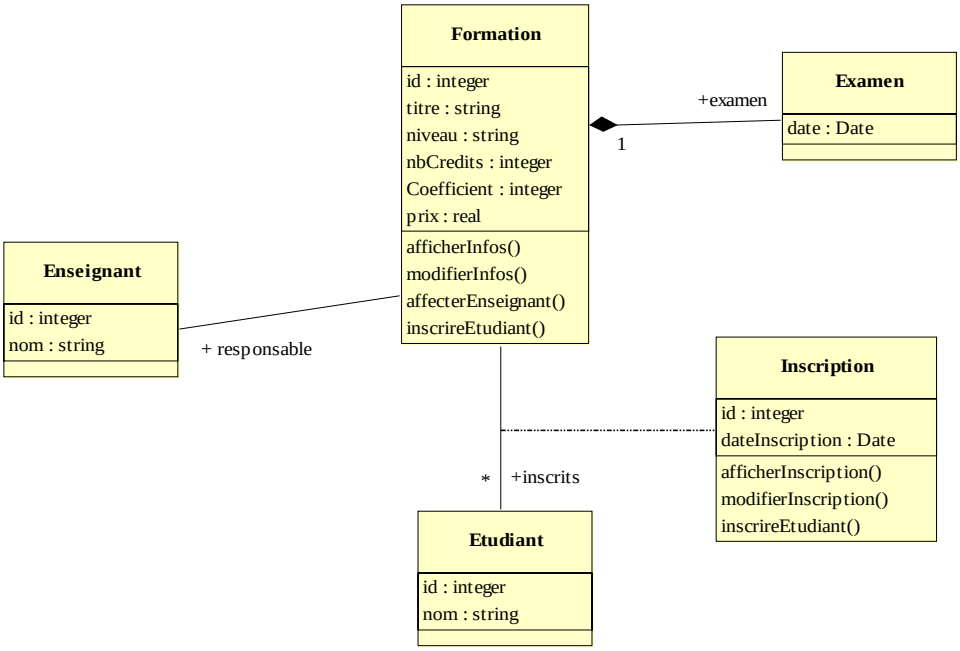

Figure 110 – *Diagramme de classes - Point de vue Responsable Site*

Fusion et modélisation VUML

Comme nous l'avons vu dans la section V.3.3, la phase de la fusion permet d'identifier les classes multivues et de réaliser un diagramme de classes global permettant de produire le modèle VUML. Ce dernier doit normalement être généré d'une manière semi-automatique par une transformation de modèles qui prend en entrée tous les PIM élaborés dans la phase de spécification par point de vue et fournit en sortie un modèle PIM VUML. Pour notre exemple, les classes *Formation*, *Question*, *Reponse*, *Exercice*, *Correction*, *Examen*, *Documentation*, *Enseignant*, *Inscription*, et *Etudiant* donnent lieu à des classes multivues. En effet, ces classes apparaissent dans au moins deux PIM par point de vue. La figure 111 ci-après illustre un diagramme VUML en faisant apparaître les stéréotypes <<*base*>>, <<*view*>>, <<*viewDependency*>> et <<*viewExtension*>>. Afin de simplifier le diagramme, seules certaines classes ont été modélisées. Ainsi, les classes multivues qui ne sont pas éclatées en vues sont représentées par des classes iconifiées stéréotypées par <<*multiViewsClass*>>.

La relation de dépendance stéréotypée <<*viewDependency*>> entre les vues de l'Enseignant et de l'Etudiant, annotée par une contrainte en OCL, exprime le fait que la liste des questions posées par un étudiant relativement à une formation doit être incluse dans la liste des questions posées sur la même formation avec l'ensemble des étudiants inscrits dans cette formation. Les relations *viewDependency* permettent d'exprimer des contraintes de cohérence sur les associations comme dans notre cas ici mais aussi des contraintes liées aux attributs et aux méthodes.

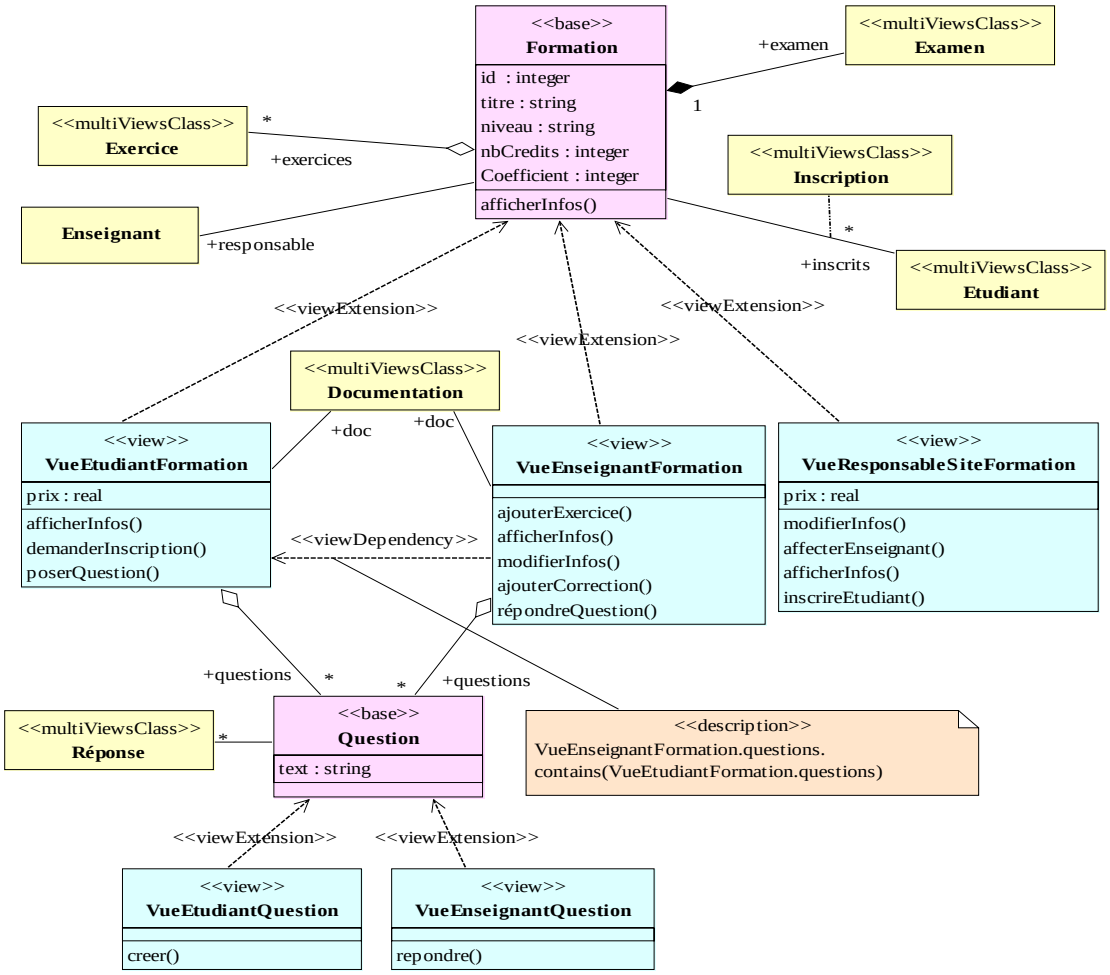

Figure 111 – *Diagramme VUML avec les classes multivues Formation et Question*

Ajout d'un point de vue

Nous avons vu dans la section V.3.4. qu'il est possible d'ajouter un point de vue à un PIM VUML. Nous avons aussi vu que, au sens MDA, cet ajout de point de vue peut être effectué par une transformation de modèles prenant en entrée un modèle PIM VUML et un modèle PIM par point de vue et donnant en sortie un modèle PIM VUML. Nous avons dit que cette transformation peut être réalisée en exploitant le mécanisme de spécialisation. Par exemple, nous nous plaçons dans le cas où une entreprise finance pour ses employés une formation du SED appelée alors *formation continue*. Le concept de *formation continue* est différent de celui de *formation* vu précédemment puisqu'il comporte notamment la gestion de dates de formation. Il fait aussi intervenir d'autres acteurs tels que les professionnels et les entreprises. Elle a bien sûr des informations communes avec une formation classique comme la documentation, les exercices, etc. On modélise cette situation en spécialisant la classe multivues *Formation* en une autre classe multivues : *FormationContinue*. Le diagramme de classes présenté dans la figure 112 illustre à titre d'exemple cette notion d'héritage en VUML. Les différents attributs, méthodes et associations ne sont pas représentés dans leur intégralité. La classe multivues *FormationContinue* hérite des vues de sa classe mère. Ainsi, il existe pour cette classe une vue enseignant permettant d'effectuer les mises à jour des exercices, de la documentation, etc.

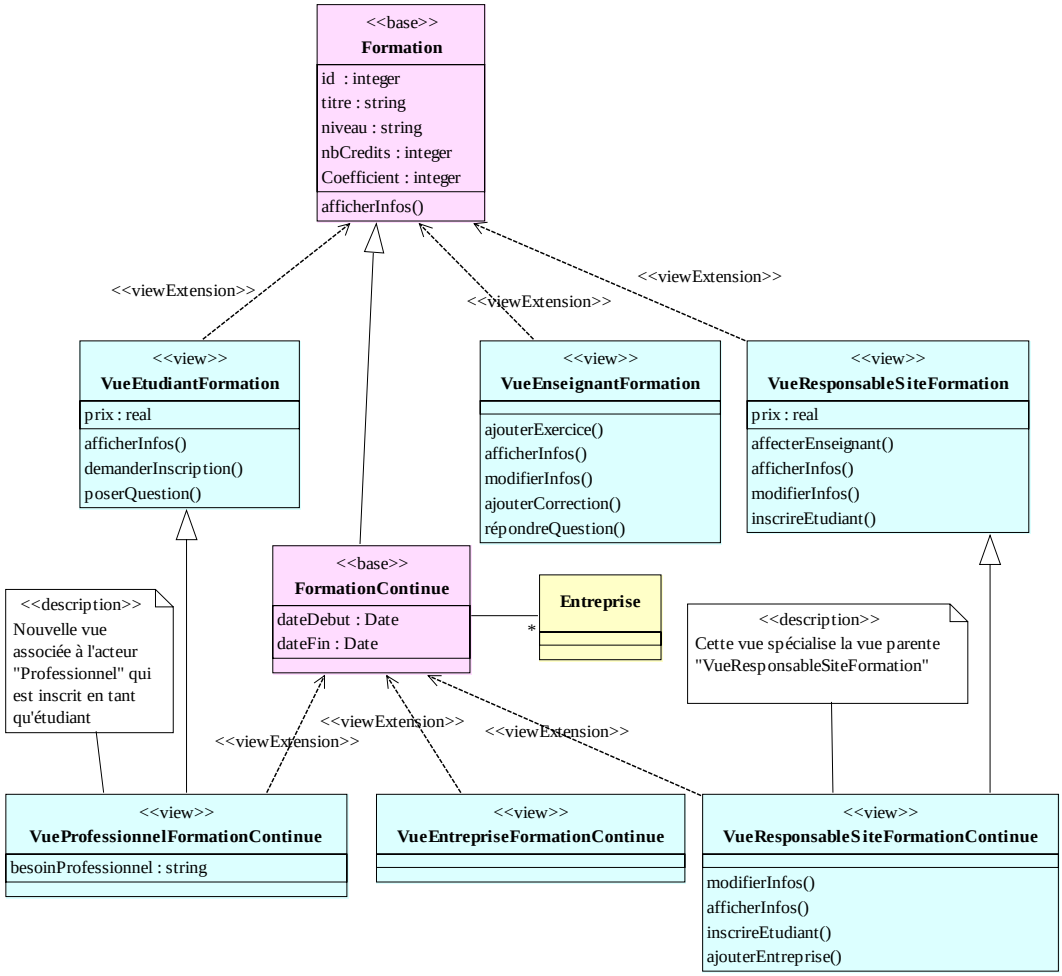

Figure 112 – *Illustration de la spécialisation de la classe multivues Formation*

Il apparaît donc judicieux de créer deux nouveaux points de vue : celui du *Professionnel* qui suivra une formation continue et celui de l'*Entreprise* dont le professionnel fait partie. Les différences entre les informations concernant un étudiant classique et celles d'un professionnel (tel que le besoin professionnel) justifient le fait de spécifier une nouvelle vue pour une formation continue : *VueProfessionnelFormationContinue*. Celle-ci hérite en la redéfinissant de *VueEtudiantFormation* puisque le professionnel — cas particulier d'étudiant — accède aux attributs et méthodes de la vue de l'étudiant comme la consultation de la documentation, la possibilité de poser des questions, etc. De même la vue du responsable du site est redéfinie en *VueResponsableSiteFormationContinue* car un ensemble d'attributs et de méthodes sont rajoutés ou redéfinis (comme *inscrireEtudiant()* qui permet d'inscrire un professionnel étudiant).

Prise en compte des plates-formes d'exécution

La dernière phase de la démarche associée à VUML consiste à appliquer des transformations de modèles (PIM->PSM) afin de prendre en compte les plates-formes d'exécution. Pour l'instant seules les plates-formes orientées objet sont prises en considération. Ainsi, nous pouvons générer le modèle objet associé au modèle VUML de la figure 111. Ce modèle objet sera utilisé par la suite pour produire le code associé à l'application.

V.5. Conclusion

Dans ce chapitre, nous avons mis l'accent sur une démarche associée à VUML dirigée par les modèles. Après une présentation synthétique de l'approche MDA, nous avons décrit les grandes lignes de la démarche associée à VUML. Ensuite, nous avons illustré cette démarche à travers une étude de cas qui concerne un système d'enseignement à distance.

Cependant un travail significatif reste, néanmoins, à effectuer afin de profiter des avantages d'une démarche dirigée par les modèles. La démarche présentée dans ce chapitre est compatible avec l'approche MDA mais elle constitue une étude préliminaire qui doit être approfondie. Il faut aussi automatiser les différentes transformations de modèles (CIM->PIM, PIM-> PIM VUML, PIM->PSM). De plus, pour générer automatiquement le code fonctionnel, il faut enrichir les modèles PIM avec un langage d'action. Ces perspectives sont les points de départ d'une thèse qui commencera en septembre 2005.

Conclusion générale

Il ne fait aucun doute que les notions de vues et de points de vue offrent un grand intérêt pour le développement de systèmes complexes, qu'ils soient entièrement logiciels ou non, depuis les phases amont telles que l'analyse des besoins jusqu'à la conception détaillée et la programmation. L'apport du concept de point de vue apparaît dans la réduction de la complexité du développement grâce à une approche décentralisée, l'amélioration de la réutilisabilité, la gestion de la cohérence des données, la prise en compte des droits d'accès selon les profils des utilisateurs, l'augmentation de l'intelligibilité de code et la réduction de la phase de test.

Les concepts de vue/point de vue ont ainsi été étudiés dans la plupart des domaines de l'informatique : bases de données, représentation des connaissances, analyse et conception, langages de programmation, outils de Génie logiciel, etc. Dans ce cadre, plusieurs approches ont été menées pour intégrer les vues/points de vue. Ces approches n'emploient pas forcément les termes de point de vue ou de vue mais peuvent introduire des termes sémantiquement proches : rôle, sujet, aspect, etc.

Le travail présenté dans cette thèse s'inscrit dans la continuité des travaux effectués dans le cadre du projet VBOOM. L'objectif global est de définir une méthode d'analyse et de conception intégrant la notion de vue/point de vue dans un cadre de modélisation par objets des systèmes complexes.

Après avoir étudié un certain nombre d'approches intégrant la notion de vue/point de vue, nous avons pu constater leurs limites pour la modélisation des systèmes complexes. Ainsi, avons-nous décidé de proposer une nouvelle approche, appelée VUML, pour l'analyse et la conception multivues.

Apport du travail

VUML offre un langage de modélisation (extension d'UML) et un noyau de démarche dirigée par les modèles. Afin de permettre une modélisation à base de vues/points de vue, VUML introduit un nouveau concept appelé "classe multivues". Une classe multivues est une entité de modélisation qui permet de décrire l'information en fonction des points de vue des acteurs concernés. Elle est statiquement composée d'une base (partie partagée par les acteurs de la classe multivues) et d'un ensemble de vues étendant cette base. Chaque vue encapsule les informations spécifiques à un acteur donné. Ce concept de classe multivues s'intègre parfaitement dans le principe de hiérarchisation des classes ; en effet, le mécanisme de spécialisation/généralisation n'a pas été modifié dans VUML : une classe multivues peut avoir des sous-classes qui sont elles-mêmes multivues. Les vues d'une classe multivues peuvent naturellement être dépendantes. La gestion de ces dépendances entre les vues est exprimée dès la phase de conception à travers des déclarations de dépendances (explicitées en OCL).

De plus, VUML introduit la notion de composant multivues qui permet de représenter une classe multivues dans le diagramme de composants. Ceci permet de profiter du principe de réutilisabilité et de configuration offert par les composants tout en conservant les avantages d'une modélisation par vues.

Afin de concrétiser notre approche et ne pas pénaliser le programmeur objet "classique", VUML propose un patron générique d'implémentation qui décrit la génération de code objet standard (Java,

C++, Eiffel...) à partir d'une modélisation VUML. Ce patron tient compte de tous les concepts et mécanismes introduits par VUML (classe multivues, héritage, dépendances entre les vues, changement dynamique de point de vue, …). Afin de gérer les droits d'accès aux services offerts par une classe multivues, le patron proposé utilise le polymorphisme et un mécanisme d'aiguillage des appels ; ceci confère à une classe multivues un dynamisme permettant d'adapter son comportement à la vue active.

En ce qui concerne la sémantique, VUML étend le métamodèle UML en introduisant un certain nombre de stéréotypes. Ces derniers sont regroupés sous forme d'un profil UML. L'utilisation de ces stéréotypes obéit à un ensemble de règles de bonne modélisation (well-formedness rules). Ainsi, la sémantique statique de VUML est décrite par le métamodèle, des règles de bonne modélisation exprimés d'une manière formelle en OCL (Object Constraint Language) et des descriptions textuelles précises. La sémantique dynamique quant à elle est décrite de façon informelle.

Concernant le processus de développement supporté par VUML, nous proposons, en continuité avec la méthode VBOOM, un noyau d'une démarche supportant la réalisation décentralisée des modèles et leur mise en cohérence de la phase d'analyse à celle d'implantation. L'explicitation des relations de cohérence entre vues dépendantes que nous préconisons pour cela est une originalité dans le panorama des propositions existantes. La démarche proposée est une démarche dirigée par les modèles qui s'inscrit dans le cadre du MDA (Model Driven Architecture).

Dans l'objectif d'appliquer efficacement notre approche, nous avons développé un outil support à VUML. Cet outil a été implémenté en personnalisant l'atelier *Objecteering/UML* grâce à la technique des profils. Il supporte complètement la modélisation basée sur les vues et points de vue. Il offre également deux fonctionnalités importantes : la première consiste en un vérificateur de la cohérence des modèles VUML qui doivent être conformes à la sémantique VUML ; la deuxième concerne la génération de code objet standard à partir d'une modélisation VUML. Cette génération de code s'appuie sur le patron d'implémentation générique proposé par VUML.

L'expérimentation de l'approche VUML pour modéliser différents systèmes (Concessionnaire de voitures, Système d'Enseignement à Distance (SED), Système de prévention de la pollution marine au MAROC (Marzak et al., 2002)) a été concluante.

Comparaison de VUML avec les approches similaires

Dans cette section nous nous intéressons aux approches qui nous semblent les plus proches de l'approche VUML et pour mieux situer l'apport de notre travail, nous proposons une comparaison synthétique de VUML avec les approches similaires présentées dans l'état de l'art (chapitre I).

Commençons tout d'abord par la norme IEEE 1471. En effet, la prise en compte des points de vue lors des phases de définition de l'architecture d'un système a donné naissance à cette norme (Hilliard, 2000). Les principes de cette recommandation rejoignent ceux que nous avons adoptés dans VUML (correspondance unique entre un point de vue et une vue) mais cette norme se situe à un niveau de granularité large, ne propose pas de langage de modélisation particulier, et n'offre pas de guidage sur la manière de trouver les points de vue. Dans VUML, nous intégrons la notion de vue à un niveau de granularité variable qui va jusqu'au niveau fin de la classe, et un point de vue correspond systématiquement aux besoins d'un acteur unique ce qui facilite la phase d'analyse.

Dans le domaine des bases de données (Abiteboul et al. 1991, Debrauwer 1998), la notion de vue existe depuis longtemps. Les vues sont exploitées par les langages d'interrogation comme des

fonctions de sélection sur les données, ou par des fonction d'IHM. Avec VUML, nous avons l'ambition de construire les vues dès l'analyse et de conserver cette notion de vue jusqu'au code. En fait, ces deux approches ne sont pas exclusives car elles visent des objectifs complémentaires.

Harrison et Ossher (Harrison et al., 1993) proposent la programmation orientée sujet comme une façon de construire des applications intégrées multi-perspectives. Les rôles (Anderson et al., 1992) et la modélisation de rôles (Kristensen et al. 1996, Riehle et al. 1998, Gottlob et al. 1996, VanHilst et al. 1996) ont été proposés pour exprimer et abstraire les interactions et les évolutions d'objets. Ces deux approches nous paraissent intéressantes pour la modélisation statique lors de la phase de conception. Par contre, elles ne sont pas particulièrement adaptées pour la phase d'analyse où il s'agit de traiter les besoins des acteurs.

La programmation par aspect introduite dans (Kiczales et al., 1997) vise la modélisation d'aspects orthogonaux d'un système. Les aspects non fonctionnels (performance, sécurité, optimisation, persistance, etc.) peuvent être ajoutés séparément au modèle fonctionnel. Cette approche ne nous paraît pas au même niveau que VUML dans la mesure où les points de vue des acteurs ne peuvent pas être considérés comme des aspects orthogonaux puisqu'ils sont en premier lieu fonctionnels. Cependant, la démarche VUML et notamment la fusion de modèles mériterait d'être reconsidérée dans un contexte d'aspects fonctionnels au niveau de l'analyse/conception (AOSD).

Les travaux de Mili et al. (Mili et al., 2000) se situent dans la programmation par vues. Ils considèrent qu'un objet peut être décrit comme une base et un ensemble de vues représentant les facettes qui peuvent lui être ajoutées ou retirées dynamiquement, exprimant ainsi ses changements de rôles potentiels. Sur ce point, l'approche de Mili et al. est similaire à celle adoptée dans VBOOM. Au contraire, VUML a opté pour une approche associant un point de vue et donc une vue unique à chaque type d'acteur. Ce choix permet de simplifier la phase d'analyse en supprimant notamment la subjectivité inhérente à la définition des vues, et en facilitant par la suite l'ajout/suppression de points de vue.

Au niveau des langages de programmation par objet, les notions de vues ont également été largement expérimentées en liaison avec les mécanismes de délégation ou d'héritage (voir par exemple Carré et al. 1991, Marcaillou 1995) sans que ne se dégage de solution globale implantée dans les principaux langages à objet du marché. Contrairement à l'approche fondée sur le langage dédié VBOOL (Marcaillou et al., 1994), nous avons choisi avec VUML de ne pas développer un nouveau formalisme ad hoc, mais d'élaborer un profil VUML. Nous fournissons par contre un patron d'implantation générique permettant, à partir d'un modèle VUML, d'engendrer facilement du code objet dans divers langages cibles (Java, C++, Eiffel).

Bardou (Bardou, 1998) propose une approche basée sur les objets morcelés pour intégrer la notion de point de vue dans les langages à prototypes. Dans cette approche la notion de point de vue correspond à une combinaison de morceaux organisés au sein d'une hiérarchie de délégation. Ceci rejoint l'approche VBOOM dans laquelle un point de vue est une combinaison de vues même si VBOOL est basé sur l'héritage multiple. Le travail de Bardou offre bien une solution permettant d'implémenter la notion de point de vue, mais, contrairement à VUML, il ne présente pas de procédé pour élaborer les points de vues pertinents d'un système.

La nouvelle version UML 2.0 (OMG, 2003b) ne propose pas de changement par rapport à la notion actuelle de vue dans UML 1.5 (UML, 2003a). La vue dans UML est un moyen offert au concepteur pour structurer l'analyse/conception en fonction de la phase de développement (vue des cas d'utilisation, vue logique, vue des composants, vue du déploiement). Quand le diagramme de classes

est réalisé, il n'y a plus de trace au niveau des classes des besoins et des droits d'accès spécifiques des acteurs du système, ce qui présente une lacune majeure par rapport à notre approche. Par ailleurs l'exploitation d'interfaces multiples en UML ne donne pas satisfaction dans la mesure où elle présente le même inconvénient que l'héritage multiple pour représenter l'évolution des points de vue et l'ajout de vues à une classe multivues (Björkander et al. 2003). L'utilisation du stéréotype "*view*" dans l'outil IBM-Rose (Rose-IBM, 2004) correspond quant à lui à l'affichage de données selon le classique modèle MVC (Model, View, Controller). Il ne permet pas de décrire une classe multivues.

Concernant les propositions à base de composants de type UML qui supporte explicitement les notions de vue et de point de vue, nous citons les travaux de Muller et al. (Muller et al., 2003). Ces travaux ont repris l'approche CROME (Vanwormhoudt, 1999) comme point de départ. Cette approche vise surtout l'amélioration de la réutilisabilité des composants en combinant la notion de composant et la notion de vue. La différence par rapport à VUML réside sutout dans le fait que l'approche de Muller et al. ne propose pas de démarche pour élaborer les composants. Le modèle de composant CCM (OMG, 2002c) étend le modèle objet traditionnel de CORBA. Il permet de décrire un composant CORBA par la description de ses facettes (interfaces fournies), réceptacles (interfaces requises), événements puits (événements consommés), événements sources (événements produits) et ses attributs. Ce modèle pourrait être utilisé pour implémenter une classe multivues à l'instar de ce que nous avons fait avec les composants UML présentés en section II.4.

En ce qui concerne les méthodes d'analyse/conception par objet, il n'y a pas de méthodologie reconnue capable de supporter complètement la notion de point de vue. VUML propose une démarche dirigée par les modèles supportant la réalisation décentralisée des modèles et leur mise en cohérence de la phase d'analyse à celle d'implantation.

Travaux en cours et futurs

Certains éléments concernant le langage de modélisation de VUML nécessitent encore des approfondissements. Comme nous l'avons déjà dit, les diagrammes dynamiques VUML qui dépendent d'un seul point de vue (tels que les diagrammes de séquence et les diagrammes de collaboration) sont similaires à ceux d'UML. Par contre les diagrammes dynamiques qui dépendent de plusieurs points de vue (tels que les diagrammes d'activité et les diagrammes d'état-transition) sont modifiés par rapport à leurs homologues dans UML. Nous travaillons actuellement à l'intégration de la notion de point de vue au sein de ces diagrammes.

Concernant l'outil support à VUML, nous sommes en train d'enrichir le générateur de code afin qu'il prenne en compte les dépendances fonctionnelles entre les vues et la propagation de la vue active. Ce générateur de code cible pour l'instant le langage Java. Afin de cibler d'autres langages à objets (C++, Eiffel, …), nous travaillons actuellement au développement d'autres profils de génération de code en réutilisant les techniques employées dans le développement du profil de génération de code Java.

Nous envisageons très prochainement de formaliser la sémantique dynamique de VUML. Cette formalisation va contribuer à résoudre les problèmes liés aux mauvaises interprétations. Dans ce contexte et à l'instar des travaux menés par le groupe pUML (the precise UML Group), une des perspectives de ce travail est de réaliser une définition plus rigoureuse de la sémantique de VUML.

En ce qui concerne la démarche de développement supportée par VUML, nous avons proposé un noyau d'une démarche dirigée par les modèles qui décrit les grandes lignes de cette démarche. Un travail important reste néanmoins à fournir en particulier dans la phase de fusion pour fonder celle-ci sur des bases mathématiques solides étayées par des propriétés prouvées et des algorithmes validés. Le travail sur la fusion consiste aussi à spécifier et implanter les transformations de niveau PIM permettant de passer de modèles UML par point de vue à un modèle VUML. Ce travail va permettre l'outillage de la démarche associée à VUML en utilisant des ateliers tel que Softeam MDA Modeler.

Enfin, des travaux de recherche sont en cours afin de prendre en compte la distribution des composants multivues (El Asri, 2005). D'autre part et afin de favoriser la réutilisation de composants multivues, l'élaboration d'un langage (ontologie) de patrons multivues sera abordée dans une thèse qui commencera en septembre 2005.

Publications

Cette section regroupe par ordre chronologique inverse, les publications que nous avons produites pendant cette thèse.

Nassar M., Coulette B., Guiochet J., Ebersold S., El Asri B., Crégut X., Kriouile A., "Vers un profil UML pour la conception de composants multivues", article soumis à la revue L'Objet (deuxième relecture), juin 2005.

El Asri B., Nassar M., Coulette B., Kriouile A., "Assemblage de composants multivues par contrats", Actes du XXIII^{ème} Congrès INFORSID (INFORSID'2005), Grenoble, France, 24-27 mai, 2005. pp. 29-44.

El Asri B., Nassar M., Coulette B., Kriouile A., "MultiViews component for information development", Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS'2005), Miami, USA, May 24-28, 2005. pp. 217-225.

Crégut X., Marcaillou S., Nassar M., Coulette B., "Un patron de génération de code pour le profil VUML", LMO-OCM'2005, Berne, Suisse, 9-11 mars 2005. pp. 5-11.

Nassar M., Coulette B. et Kriouile A., "Génération de code dans VUML". Journal Marocain d'Automatique, d'Informatique et de Traitement du Signal, article sélectionné de la conférence COPSTIC'03. 2004. pp. 87-97.

Nassar M., El Asri B., Coulette B. et Kriouile A., "Une approche UML de composants multivues". Workshop Objets-Composants-Modèles dans les Systèmes d'Information (OCM-SI'2004). Biarritz, France. 25 mai 2004. pp. 17-24.

El Asri B., Nassar M., Coulette B., Kriouile A., "Views, Subjects, Roles and Aspects: A comparison along Software Lifecycle", Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS'2004), Porto, April 14-17, 2004. pp. 139-146.

Nassar M., "VUML : a Viewpoint oriented UML Extension", Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'2003) - Symposium doctoral, Montreal, Canada, Octobre 06-10, 2003. pp. 373-376.

Nassar M., Coulette B., Kriouile A., “Vers des composants multivues réutilisables”. Workshop Objets, Composants et Modèles dans l’ingénierie des Systèmes d’Information (OCM-SI’2003), Nancy, France, 3 juin 2003. pp. 49-54.

Nassar M., Coulette B., Crégut X., Marcaillou S., Kriouile A., “Towards a View based Unified Modeling Language”, Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS’2003), Angers, France, April 23-26 2003. pp. 257-265.

Nassar M., Coulette B., Kriouile A., “Vers un langage de modélisation unifié supportant les vues”, Rapport IRIT 02-29-R, Toulouse, France. Novembre 2002.

Bibliographie

Abiteboul S., Bonner A., "Objects and Views", *Proceedings of ACM SIGMOD*, mai 1991, pp. 238-247.

Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et réparti), "La démarche MDA", Livrable 1.1-5, Mai 2002

Andersen E. P., Reenskaug, "System Design by Composing Structures of Interacting Objects", *Proc. of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, LNCS, Vol. 615. pp. 133-152, Utrecht, The Netherlands. Springer-Verlag. 1992.

Andersen E. P., "Conceptual Modeling of Objects. A Role Modeling Approach", PhD thesis, Department of Informatics, University of Oslo ,November 1997.

ATL. ATL Development Tools. INRIA, LINA et Université de Nantes, December 2004. http://www.sciences.univ-nantes.fr/lina/atl

Bardou D., "Etude de langages à prototypes, du mécanisme de délégation et de son rapport à la notion de point de vues", thèse de doctorat en Informatique, LIRMM, université de Montpellier 2, 1998.

Bardou D., et al., "Roles, Subjects and Aspects: How do they relate?", *Position paper at the Aspect Oriented Programming Workshop. 12th European Conference on Object-Oriented Programming (ECOOP '98), LNCS*, vol. 1543, Springer. 1998.

Bendelloul S., Mili H., Dargham J., Mcheick H., "A comparison of view programming, aspect-oriented programming, subject-oriented programming from a reuse perspective", *Proc. of 13th ICSSEA*, Volume 4, Paris, France, 2000.

Bézivin J., Dupé G., Jouault F., Pitette G. et Rougui J., "Firt Experiments with the ATL Model Transformation Languages: Transforming XSLT into XQuery". *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

Bézivin J., Blay M., Bouzeghoub M., Estublier J., Favre J. M., Rapport de synthèse, Action Spécifique CNRS sur le MDA, janvier 2005.
http://www.planetmde.org/as/rapport/AS-MDA-IDM-Synthese-1.1.pdf

Björkander M, Kobryn C., "Architecturing Systems with UML 2.0", *IEEE software*, juillet/aout 2003.

Blanc X., *MDA en action*, Edition Eyrolles, 2005.

Bobrow D.G., Stefik M., "The LOOPS Manual : a Data and Object-Oriented Programming System for Interlisp", *Knowledge-Based VLSI Design Group, Memo KB-VLSI-81, Xerox PARC*, Palo Alto, California, 1983.

Booch G., "Object Oriented Design with applications", *The Benjamin/Cummings publishing company*, Inc, 1991.

Booch G., "Object Oriented Analysis and Design with Applications", *2nd edition The Benjamin/Cummings publishing company*, Redwood City, CA, 1994.

Booch G., Brown A., Iyengar S., Rumbaugh J., Selic B., The IBM MDA Manifesto The MDA Journal, May 2004, http://www.bptrends.com

Bruel J-M., France R., "Transforming UML models to formal specifications", *Proceedings of UML'98 – Beyond the notation, LNCS. Springer*, June, 1998.

Carré B., "Méthodologie orientée objet pour la représentation des connaissances, concepts de points de vue, de représentation multiple et évolutive d'objets", Thèse du LIFL, 1989.

Carré B., L. Dekker, Geib J.M., "Multiple and Evolutive Représentation in the ROME language", *actes de "TOOLS'90"*, 1990a, pp. 101-109.

Carré B., Geib J.M., "The Point of View Notion for Multiple Inheritance", *Proceedings of ECOOP/OOPSLA'90*, 1990b, pp. 312-321.

Carré B., Dekker L., "Inheriting Object-Oriented Features through Meta-Programming – A Frame Extension to ROME", *actes de "East Europe'91 Conference on Object-Oriented Programming"*, Bratislavia(Cz), 15-19 septembre 1991.

Charrel P.J., Galaretta D., Hanachi C., Keller P., Rothenburger B., "Multiple viewpoints for the design of complex space systems", *actes de "4th workshop on Artificial Intelligence and knowledge-Based Systems for Space"*, ESTEC, Noordwijk (NI), 17-19 mai 1993a, pp. 251-272.

Charrel P.J., Galaretta D., Hanachi C., Rothenburger B., "Multiple Viewpoints for Development of Complex Software", *actes de IEEE International Conference on Systems, Man and Cybernetics*, 17-20 octobre 1993b, pp. 556-561.

Charrel P.J., Galaretta D., Rothenburger B., "An Approach to Designing Based on a Multi-viewpoint Confrontation of Specular Agents", *actes de "4th European-Japanese Seminar on Information Modelling and Knowledge Bases"*, Stockholm(S), 31 mai- 06 juin 1994.

Charrel P.J., "Points de vue et représentations", HDR de l'université Toulouse 1, 2000.

Clarke S., Harrison W., Ossher H., Tarr P., "Separating Concerns throughout the Development Lifecycle". *Proc. Of ECOOP'99 Workshop on Aspect-Oriented Programming*, Lisbon, 1999.

Coad P., Yourdon E., *Object Oriented Design*, Prentice-Hall international editions, NJ, 1991.

Coad P., "Object-oriented Patterns", *Communications of the ACM*, 1992.

Cook S., "Domain-Specific Modeling and Model Driven Architecture", *MDA Journal*, January 2004, pp. 1-10.

Coulette B., Kriouile A., Marcaillou S., "L'approche par points de vue dans le développement orienté objet des systèmes complexes", *Revue l'Objet*, vol. 2, n°4, février 1996, pp. 13-20.

Coulette B. et al. *Réseau STIC franco-marocain en Génie Logiciel*, 2002. http://www.univ-tlse2.fr/grimm/isycom/reseauSTIC/reseauSTIC.html

Crégut X., Marcaillou S., Nassar M., Coulette B., "Un patron de génération de code pour le profil VUML", *LMO-OCM'2005*, Berne, Suisse, 9-11 mars 2005. pp. 5-11.

Cueignet X., Lextrait V., "Génération de serveur de vues", Thèse de l'université de Sophia Antipolis, décembre 1992.

Czarnecki K., et Helsen S., "Classification of Model Transformation Approaches", *proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the context of MDA*, October 2003.

Debrauwer L., "Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME", Thèse de doctorat en Informatique, LIFL, Université des Sciences et Technologies de Lille, décembre 1998.

Dekker L., Carré B., "Multiple and dynamic representation of frames with points of view in FROME", *actes de "Représentation Par Objets"*, La Grande Motte, 17-18 juin 1992, pp. 97-111.

Dekker L., "La réification des filtres en FROME", *actes de "PRO"*, La Grande Motte, 17-18 juin 92, pp. 23-35.

Dekker L., "FROME : Représentation multiple et classification d'objets avec points de vue", thèse de l'Université de Lille, juin 1994.

Desfray P., *Ingénierie des objets : Approche Classe-Relation application à C++*, Masson, 1992.

Dhiba Y., "Intégration des aspects coopératifs dans la phase de conception de la méthode orientée-objets multi-vues VBOOM", thèse pour l'obtention du diplôme de spécialité de 3ème cycle de l'université Mohamed V, 1999.

Ducourneau R., Habib M., "La multiplicité de l'héritage dans les langages à objets", *Technique et Science Informatique*, AFCET- Bordas 1989, pp. 41-62.

Dugerdil P., "Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG", thèse de l'université d'Aix-Marseille II, décembre 1988.

DSTC, IBM et CBOP, "MOF Query / Views / Transformations – Second Revised Submission", ad/2004-01-06, January 2004.

El Asri B., Kriouile A., "Fusion de modèles visuels dans la méthode VBOOM", Rapport IRIT/96-39-R, Toulouse, octobre 1996.

El Asri B., Kriouile A., Boulmakoul A., Coulette B., "Application de l'approche objet orientée point de vue à la modélisation des Interactions Transport-Environnement", *actes du 4ème Colloque Africain sur la Recherche en Informatique (CARI'98)*, Dakar (Sénégal), 12-15 octobre 1998, pp. 497-508.

El Asri B., Nassar M., Kriouile A., Coulette B., "Views, subjects, roles and aspects : A comparison along software lifecycle", *Proceedings of 6th International Conference on Enterprise Information Systems ICEIS'04*, Porto-Portugal, 14-17 April 2004.

El Asri B., Nassar M., Coulette B., Kriouile A., "Assemblage de composants multivues par contrats", Actes du XXIIIème Congrès INFORSID (INFORSID'2005), Grenoble, France, 24-27 mai, 2005a. pp. 29-44.

El Asri B., Nassar M., Coulette B., Kriouile A., "MultiViews component for information development", Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS'2005), Miami, USA, May 24-28, 2005b. pp. 217-225.

El Asri B., "Vers des composants multivues distribués", Thèse nationale en cours à l'ENSIAS de Rabat (soutenance prévue en octobre 2005).

Elrad T., Aksits M., Kiszales G., Lieberher K., Ossher H., "Discussing Aspects of AOP", *Communications of the ACM*, vol. 44, no. 10, October 2001, pp 33-38.

Evans A., France R., Lano K., Rumpe B., "The UML as a formal modelling notation", *Proceedings of UML'98 – Beyond the notation, LNCS. Springer*, June, 1998.

Evans A., Kent S., "Meta-modelling semantics of UML: the pUML approach", *Proceedings of UML'99 – Beyond the Standard*, Fort Collins, USA, October 1999, Lecture Notes in Computer Science vol. 1793, pp 141-155, Springer, 1999.

Favre J. M., "Towards a Basic Theory to Model Driven Engineering", *UML 2004, Workshop in Software Model Engineering (WISME 2004)*, 2004.

Finkelstein A., Kramer J., Goedicke M., "Viewpoint Oriented Software Development", Proceedings of Software Engineering and Applications Conference, Toulouse, December 1990, p. 337-351.

Finkelstein A., Gabbay D., Hunter A., Kramer J., Nuseibeh B., "Inconsistency Handling in Multi-Perspective Specifications", *actes de conférence ESEC'93*, Garmish-Paternkirchen (D), septembre 1993, pp. 84-99.

Fowler M., "Application views : Another technique in the analysis and design armoury", *Revue JOOP*, 03 avril 1994, pp. 59-66.

France R., "A Problem-Oriented Analysis of basic UML Static Requirements Modelling Concepts", *OOPSLA'99*, ACM SIGPLAN Notices, Vol. 34, No. 10, October 1999, pp. 57-69.

Gamma E. et al., *Design Patterns, Elements of reusable Object-oriented Software*, Addison-Wesley, 1995.

Goldstein I.P., Bobrow D.G., "Extending Object Oriented Programming in Smalltalk", *actes de "Lisp Conference Standford"*, 1980, pp. 75-81.

Grønmo R., Belaunde M., Aagedal J, Engel K., Faugere M., Solheim I., "Evaluation of the Proposed QVTMerge Language for Model Transformations", *Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Systems (WSMDEIS 2005), In conjunction with ICEIS 2005*, Miami, U.S.A., May 2005.

Gottlob G., Schrefl M., Roeck B., "Extending object-oriented systems with roles", *In ACM Transactions on Information Systems*, vol. 14 n. 3, pp. 268-296, 1996.

GreenField J., Short K., with Cook S., Kent S., (forword by Crupi J.) – Software Factories, Assembling Applications with Patterns, Models, Frameworks and Tools, Wiley Publishing, 2004.

Habermann A.N., Krueger C., Pierce B., Staudit B., Wenn J., "Programming with views", Rapport interne CMU-CS-87-177, 29/01/1988.

Hanenberg S., Unland R., "Roles and Aspects: Similarities, Differences, and Synergetic Potential", OOIS'2002, Springer-Verlag Berlin Heidelberg 2002, pp. 507-520.

Harrison W., Ossher H., "Subject-oriented programming : a critique of pure objects", *Proceedings of OOPSLA'93*, Washington D.C., Sept. 26-Oct 1, 1993, pp. 411-428.

Hair A., "Conception de VBTOOL, outil support de la méthode VBOOM, réalisation des fonctionnalités : Fusion et Génération du code", thèse pour l'obtention du diplôme de spécialité de 3ème cycle de l'universite Mohamed V, 1997.

Hair A., El Asri B., Kriouile A, Coulette B., "Outil support de la méthode VBOOM, Fonctionnalités Fusion et Génération du code", *actes du 4ème Colloque Africain sur la Recherche en Informatique (CARI'98)*, Dakar (Sénégal), 12-15 octobre 1998, pp. 497-508.

Hilliard R., "EEE1471-std-2000 : Recommended Practice for Architectural Description for Software-Intensive Systems", November 2000.
http://www.enterprise-architecture.info/Images/Documents/IEEE%201471-2000.pdf

Interactive Objects Software GMBH AND Project Technology, Inc.. "2nd Revised Submission to MOF Query / View / Transformation RFP", January 2004. http://www.omg.org/docs/ad/04-01-14.pdf

Jacobson I., Christerson M., Jonsson P., Overgaard G., *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, ACM Press, second edition, 1993.

Jézéquel J-M, Gérard S., Mraidha C., Baudry B., Approche unificatrice par les modèles, Action Spécifique CNRS sur le MDA, janvier 2005.

Joshi R. K., Agrawal N., "AspectJ Implementation of a Dynamically Pluggable Filter Objects in Distributed Environment", *Proceedings of 2nd Int. Wshop Aspect Oriented Soft. Dev.*, Bonn, Feb 21-22, 2002.

Kiczales G., Lampng J., Mendhekar A., Maeda C., Lopes C.V., "Aspect-Oriented Programming", *Proceddings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, Springer-Verlag LNCS 1241. June 1997.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.G., "An Overview of AspectJ", *Proceeding of ECOOP'01*, Springer Verang LNCS2072, 2001.

Kruchten P., "The 4+1 View Model of Architecture", *IEEE Software*, vol. 12, No. 6, November 1995, pp. 42-50.

Kriouile A., "VBOOM, une méthode orientée objet d'analyse et de conception par points de vue", thèse d'Etat de l'université Mohammed V de Rabat, 1995.

Kristensen B. B., "Object Oriented Modeling with Roles", *Proceedings of the 2nd International Conference on Object Oriented Information Systems (OOIS'95)*, Dublin, Irland, 1995.

Kristensen B. B., Osterbye K., "Roles: Conceptual Abstraction Theory & Practical Language Issues", *Theory and Practice of Object Systems (TAPOS)*, pp. 143-160, 1996a. Special Issue on Subjectivity in Object-Oriented Systems.

Kristensen B. B., May D.C.M., "Activities: Abstractions for Collective Behavior", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'96)*, Linz, Austria, 1996b.

Kurtev I., Bézivin J., Aksit M.., "Technological Spaces: An Initial Appraisal", *CoopIS, DOA'2002 Federated Conferences, Industrial tarck*, 2002.

Le Moigne J.L., *la modélisation des systèmes complexes*, Dunod, 1990.

Lieberman, H. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems". Dans Meyrowitz, N. K., éditeur, Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), 1986, pp. 214-223, Portland, Oregon, USA. Published as ACM SIGPLAN Notices 21(11).

Lopes D. C. P., "Etude et applications de l'approche MDA pour des plates-fromes de Service Web", Thèse de Doctorat en Informatique, LINA, Université de Nantes, juillet 2005.

Maes P., "Concepts and Experiments in Computational Reflection", Proceedings of OOPSLA'87, 1987.

Marcaillou S., Kriouile A., Coulette B., "VBOOL : une extension d'Eiffel intégrant le concept de points de vue", *actes de MCSEAI'94*, Rabat, 11-14 Avril 1994, pp. 115-125.

Marcaillou S., "Intégration de la notion de points de vue dans la modélisation par objets – Le langage VBOOL", thèse de l'université Paul Sabatier de Toulouse, 1995.

Marcaillou S., Coulette B., "Semantics of visibility, a New Relationship for View Based Object-Oriented Modelling", Rapport IRIT /96-40-R, Toulouse, octobre 1996.

Marino O., Rechenmann F., Uvietta, "Multiple Perspectives and Classification Mechanism in Object-Oriented Representation", *actes de ECAI'90*, Stockholm, 1990, pp. 425-430.

Marino O., "Classification d'objets composites dans un système de représentation de connaissances multi points e vue", *actes du 8ème congrès RFIA*, Lyon, 1991, pp. 233-242.

Marino O., "Raisonnement classificatoire dans une représentation à objets multi-points de vue", thèse de l'Université Joseph Fourier- Grenoble 1, novembre 1993.

Marzak A., "Conception de VBTOOL, outil support de la méthode VBOOM, réalisation des fonctionnalités : Analyse et conception", Thèse pour l'obtention du diplôme de spécialité de 3ème cycle de l'université Mohamed V, 1997.

Marzak A., Coulette B., "Rapport de conception du prototype DOME", Convention CNR-CNRST n° 11579, juillet 2002.

Marschall F. et Braun P., "Model Transformations fort the MDA with BOTL", *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, June 2003.

Meyer B., Eiffel the Language, Prentice Hall International (U.K), Ltd,1992.

Meyer B., *Object success - A managers's guide*, Prentice Hall - The Object-Oriented Series, 1995.

Mili H., Dargham J., "View Programming in C++: A co-reference based approach", *Rapport technique*, Département d'Informatique, Université du Québec à Montréal, Canada, décembre 1997.

Mili H, Dargham J., Mili A., "Views: A Framework for Feature-Based Development and Distribution of OO Applications", *Proceedings of the Thirty-Third Hawaii International Conference on System Sciences*. Honolulu, HI, January 4-9, 2000.

Mili H, Mcheick H., Dargham J., Dalloul S., "Distribution d'objets avec vues", *Revue L'Objet-7/2001*, LMO'2001, 2001, pp. 27-44.

Mili H., Mcheick H., Sadou S., "CorbaViews – Distributing Objects that Support Several Functional Aspects", *in Journal of Object Technology*, vol. 1, no. 3, Special issue : TOOLS USA 2002 proceedings, 2002, pp. 207-229.

Motschnig-Pitrik R., "The Viewpoint Abstraction in Object-Oriented Modeling and the UML", *International conference on Conceptual Modeling (ER 2000)* Salt Lake City, Utah, USA, 2000.

Motshnig-Pitrik R., Schett M., "Customizing web-based systems with object-oriented views" *Proceedings of 5th International Conference on Enterprise Information Systems ICEIS'03*, Angers, 23-26 April 2003.

Muller A., Caron O., Carré B., Vanwormhoudt G., "Réutilisation d'aspects fonctionnels des vues aux composants", *Revue RSTI-L'objet*, vol. 9, n°1-2, 2003, LMO'2003, pp. 241-255.

Nassar M., "Vers une programmation orientée objet par points de vue : Conception et Réalisation d'un compilateur pour le langage VBOOL", Thèse de spécialité de 3ème cycle en Informatique, université Mohammed V- Agdal, Rabat, 1999.

Nassar M., Coulette B., Kriouile A., "Vers un langage de modélisation unifié supportant les vues", Rapport IRIT 02-29-R, Toulouse, octobre 2002.

Nassar M., Coulette B., Crégut X., Marcaillou S., Kriouile A., "Towards a View based Unified Modeling Language", *Proceedings of 5th International Conference on Enterprise Information Systems ICEIS'03*, Angers, 23-26 April 2003a, pp. 257-265.

Nassar M., Coulette B., Kriouile A., "Vers des composants multivues réutilisables", *Workshop Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information (OCM-SI'2003)*, Nancy, France, 3 juin 2003b. pp. 49-54.

Nassar, M., "VUML : a Viewpoint oriented UML Extension", *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'2003 - Doctoral symposium)*. Montreal, Canada, October 6-10, 2003. pp. 373-376.

Nassar M., Coulette B. et Kriouile A., "Génération de code dans VUML". *Journal Marocain d'Automatique, d'Informatique et de Traitement du Signal*, article sélectionné de la conférence COPSTIC'03. 2004a.

Nassar M., El Asri B., Coulette B. et Kriouile A., "Une approche UML de composants multivues", *Workshop Objets-Composants-Modèles dans les Systèmes d'Information (OCM-SI'2004)*, Biarritz, France, 25 mai 2004b. pp. 17-24.

Nassar M., "VUML : une extension UML orientée point de vue", Thèse de Doctorat en Informatique, Rabat : ENSIAS, Université Mohammed V-Souissi. 2004.

Nassar M., Coulette B., Guiochet J., Ebersold S., El Asri B., Crégut X., Kriouile A., "Vers un profil UML pour la conception de composants multivues", *article soumis à la revue L'Objet (deuxième relecture)*, 2005.

NetBeans.ORG. Metadata Repository (MDR), 2003. http://mdr.netbeans.org

Nuseibeh B., Finkelstein A., Kramer J., "Method Engineering for Multi-Perspective Software Development", *Information and Software Technology Journal*, 38(4): 267-274, Elsevier Science B.V., April 1996.

QVT-MERGE GROUP, "Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10) ", April 2004. http://www.omg.org/docs/ad/04-04-01.pdf

Patrascoiu O., "YATL: Yet Another Transformation language", *in Proceedings of the 1st European MDA Workshop MDA-IA*, pp. 83-90, January 2004.

OMG, Meta Object Facility (MOF) specification – *version 1.4, formal/01-11-02*, April 2002a. http://www.omg.org/docs/formal/02-04-03.pdf

OMG, XMI, XML Metadata Interchange (XMI), v1.2, OMG, 2002b.
http://www.omg.org/docs/formal/02-01-01.pdf

OMG, *CORBA Components , version 3.0 full specification*. OMG document formal/02-06-65. June 2002c. http://www.omg.org/docs/formal/02-06-65.pdf

OMG, "Request for Proposal: MOF 2.0 Query/Views/transformations RFP", October 2002d. http://www.omg.org/docs/ad/02-04-10.pdf

OMG, Unified Modeling Language, version 1.5, 2003a.
http://www.omg.org/docs/formal/03-03-01.pdf

OMG, UML 2.0 Superstructure Final Adopted specification, Document - ptc/03-08-02, 2003b, http://www.omg.org/docs/ptc/03-08-02.pdf

OMG, UML 2 OCL Final Adopted Specification, 2003c
http://www.omg.org/docs/ptc/03-10-14.pdf

OMG, MDA Guide Version 1.0.1, Object Management Group, 2003d.
http://www.omg.org/docs/omg/03-06-01.pdf

OMG, CWM, Common Warehouse Metamodel™ (CWM™) Specification, v1.1, OMG, 2003e.
http://www.omg.org/docs/formal/03-03-02.pdf

OMG-Site, www.omg.org, 2004.

Objecteering - site, 2004, http://www.objecteering.com

Objecteering, *Objecteering/UML Modeler User Guide (version 5.3)*, 2004a
http://www.objecteering.com/pdf/doc/us/UMLModeler.pdf

Objecteering, *Objecteering/UML Profile Builder User Guide*, 2004b
http://www.objecteering.com/pdf/doc/us/UMLProfileBuilder.pdf

Objecteering, *Objecteering/Metamodel User Guide*, 2004c
http://www.objecteering.com/pdf/doc/us/Metamodel.pdf

Objecteering, *Objecteering/J Language User Guide*, 2004d
http://www.objecteering.com/pdf/doc/us/JLanguage.pdf

Ossher H., Kaplan M., Harrison W., Katz A., Kruskal V., "Subject-oriented composition rules", *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, Austin, TX, OOPSLA'1995*; Oct. 15-19 1995, pp. 235-250.

Ossher H., Tarr P., "Using multidimensional separation of concerns to (re)shape evolving software", *Communications of the ACM*, October 2001/Vol. 44, No. 10, pp. 43-50.

Peltier M., "Transformation entre un profil UML et un métamodèle MOF", *Revue l'Objet*, vol. 8, n°1-2/2002, LMO'2002, p. 25-40.

Pernici B., "Objects with roles", *Proceedings of the ACM--IEEE Conference on Office Information Systems*, Cambridge, MA, 1990.

Perrot J.F., Wolinski F., "Modélisation par objets en robotique", *Revue "Technique et science informatiques"*, volume 11 – n°1/1992, pp. 97-115.

Perrussel L., "Un outillage logique pour l'ingénierie des exigences multi-point de vue", thèse d'informatique, université Paul Sabatier, Janvier 1998.

Pryor J., Bastan N., "A Reflective Architecture for the Support of Aspect-Oriented Programming in Smalltalk", *Proceedings of the ECOOP'99 Workshop on Aspect-Oriented Programming*, 1999.

Reenskaug T., "Working with Objects : The OORAM Software Engineering Method", Englewood Cliffs: Prentice Hall, 1995.

Reenskaug T., "Working with Objects: a Three-Model Architecture for the Analysis of Information Systems", *Journal of Object Oriented Programming*, vol. 10 no. 2, 1997, pp. 22-30.

Reenskaug T., "Multi dimensional layering of business object component systems", *Position paper, Workshop Business Objects, OOPSLA '99*, 28 August 1999.

Riehl D., Gross T., "Role Model Based Framework Design and Integration", *Proceedings of the Conference on Object-Oriented Programming Systems, Language, and Application (OOPSLA '98)*. ACM press, 1998, pp. 117-133.

Riehle D., "Framework Design: A Role Modeling Approach", Ph.D. Thesis, No. 13509. Zrich, Switzerland, ETH Zrich, 2000.

Rieu D., Nguyen G.T., "Object Views for Engineering Databases", *actes de "third international conference on data and knowledge systems for manufacturing and engineering, AFCET'92"*, Lyon, mars 1992a, pp. 335-349.

Rieu D., Nguyen G.T., Escamilla J., "Méthodes et représentation de connaissances", *actes de "Représentation Par Objets"*, La Grande Motte, 17-18 juin 1992b, pp. 17-29.

Rose-IBM, UML resource center, http://www-306.ibm.com/software/rational/uml/, 2004.

Rumbaugh J., Blaha M., Premerlami W., Eddy F., Lorensen W., *Object oriented modeling and design*, Prentice Hall international editions, 1991.

Rumbaugh J., "On the horns of the modeling dilemna- Choosing among alternate modeling constructs", *Revue JOOP*, 11 décembre 1994, pp. 8-17.

Rundensteiner E. A., "Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases", *Dans Yuan, L.-Y., Editeur, Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)*, Vancouver, Canada. Morgan Kaufmann., 1992, pp. 187-198.

Softeam, "Profiles UML et langage J : Contrôlez totalement le développement d'applications avec UML", *White Paper*, 1999.

Soley et al., MDA Model Driven Architecture, by Richard Soley and the OMG Staff Strategy Group, Object Management Group White Paper, Draft 3.2 - November 27, 2000

Stefik M., Bobrow D.G., "Object-Oriented : Themes and Variations", *Revue "AI magazine"*, vol IV n4 de 1986, pp. 40-62.

Suzuki J., Yamamoto Y., "Extending UML with aspects : Aspect support in the design phase", *In Proc. of the third ECOOP Aspect-Oriented Programming Workshop*, 1999.

Taivalsaari, A. "A Critical View of Inheritance and Reusability in Object-Oriented Programming", PhD thesis, University of Jyväskylä, Finland, 1993.

Tarr P.L., Ossher H., Harrison W., Stanley M., Sutton Jr., "N Degrees of Separation : Multi-Dimensional Separation of Concerns", *International Conference on Software Engineering*, 1999, pp. 107-119.

VanHilst M. and Notkin D., "Using Role Components to Implement Collaboration-Based Designs", *Proc. of OOPSLA'96*, pp. 359-369, San Jose, CA, Oct., 6-10 1996.

Vanwormhoudt G., "CROME : un cadre de programmation par objets structurés en contextes" , thèse de doctorat en Informatique, LIFL, université des sciences et technologies de Lille, 1999.

Walden K., Nerson J-M., *Seamless Object Oriented Software Architecture - Analysis and design of reliable systems*, Prentice Hall Int. (UK), 1995.

Wolinski F., Perrot J.F., "Representation of complex objects : Multiple Facets with Part-Whole Hierarchies", *actes de "ECOOP'91"*, Genève, 1991, pp. 288-306.

Annexes

Annexe A : Grammaire du langage OCL

Cette annexe décrit la grammaire du langage OCL. La description de cette grammaire utilise la syntaxe EBNF.

```
oclFile := ( "package" packageName oclExpressions "endpackage" )+
packageName := pathName
oclExpressions := ( constraint )*
constraint := contextDeclaration ( ( "def" name? ":" letExpression* ) | ( stereotype name? ":" oclExpression))+
contextDeclaration := "context" ( operationContext | classifierContext )
classifierContext := ( name ":" name ) | name
operationContext := name "::" operationName "(" formalParameterList ")" ( ":" returnType )?
stereotype := ( "pre" | "post" | "inv" )
operationName := name | "=" | "+" | "-" | "<" | "<=" | ">=" | ">" | "/" | "*" | "<>" | "implies" | "not" | "or"
                      | "xor" | "and"
formalParameterList := ( name ":" typeSpecifier ("," name ":" typeSpecifier )*)?
typeSpecifier := simpleTypeSpecifier | collectionType
collectionType := collectionKind "(" simpleTypeSpecifier ")"
oclExpression := (letExpression* "in")? expression
returnType := typeSpecifier
expression := logicalExpression
letExpression := "let" name ( "(" formalParameterList ")" )? ( ":" typeSpecifier )? "=" expression
ifExpression := "if" expression "then" expression "else" expression "endif"
logicalExpression := relationalExpression ( logicalOperator  relationalExpression)*
relationalExpression := additiveExpression ( relationalOperator additiveExpression)?
additiveExpression := multiplicativeExpression ( addOperator multiplicativeExpression)*
multiplicativeExpression:= unaryExpression ( multiplyOperator unaryExpression)*
unaryExpression := ( unaryOperator postfixExpression) | postfixExpression
postfixExpression := primaryExpression ( ("." | "->")propertyCall )*
primaryExpression := literalCollection | literal | propertyCall | "(" expression ")" | ifExpression
propertyCallParameters := "(" ( declarator )? ( actualParameterList )? ")"
literal := string | number | enumLiteral
enumLiteral := name "::" name ( "::" name )*
simpleTypeSpecifier := pathName
literalCollection := collectionKind "{"( collectionItem ("," collectionItem )*)?"}"
collectionItem := expression (".." expression )?
propertyCall := pathName ( timeExpression )? ( qualifiers )? ( propertyCallParameters )?
qualifiers := "[" actualParameterList "]"
declarator := name ( "," name )* ( ":" simpleTypeSpecifier )?
                    ( ";" name ":" typeSpecifier "="expression)?"|"
pathName := name ( "::" name )*
timeExpression := "@" "pre"
actualParameterList := expression ("," expression)*
logicalOperator := "and" | "or" | "xor" | "implies"
collectionKind := "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator := "+" | "-"
multiplyOperator := "*" | "/"
unaryOperator := "-" | "not"
```

```
typeName :=charForNameTop charForName*
name := charForNameTop charForName*
charForNameTop := /* Characters except inhibitedChar and ["0"-"9"];  the available characters shall be
                     determined by the tool implementers ultimately.*/
charForName := /* Characters except inhibitedChar; the available characters shall be determined by the tool
                  implementers ultimately.*/
inhibitedChar := " " | "\"" | "#" | "\'" | "(" | ")" | "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "@" |
                  "[" | "\\" | "]" | "{" | "|" | "}"
number := ["0"-"9"] (["0"-"9"])*( "." ["0"-"9"] (["0"-"9"])* )?
            ( ("e" | "E") ( "+" | "-" )? ["0"-"9"](["0"-"9"])*)?
string := """(( ~["'","\","\n","\r"] )|("\\"( ["n","t","b","r","f","\\","'","\""] | ["0"-"7"] ( ["0"-"7"]
          ( ["0"-"7"] )? )?)))*"""
```

Annexe B : Fonctions génériques OCL

Afin de simplifier l'écriture des contraintes OCL présentés dans le chapitre III, nous avons défini les fonctions suivantes :

[1] La fonction *allStereotypes* retourne un ensemble contenant les stéréotypes de l'élément de modélisation courant et tous les stéréotypes des ancêtres de cet élément.

```
allStereotypes : Set(Stereotype);
allStereotypes = self.stereotype->union
(self.stereotype.generalization.parent.allStereotypes)
```

[2] La fonction *isStereotyped* détermine si l'élément de modélisation courant est stéréotypé par le nom indiqué comme argument.

```
isStereotyped (stereotypeName : String) : Boolean ;
self.stereotype.name = stereotypeName
```

[3] La fonction *isStereokinded* détermine si l'élément de modélisation est stéréotypé par le nom spécifié comme argument ou bien l'un de ses ancêtres est stéréotypé par ce nom.

```
isStereokinded (stereotypeName : String) : Boolean;
self.allStereotypes->exists (stereotype | stereotype.name = stereotypeName)
```

[4] La fonction *viewRoot* détermine la racine de l'élément de modélisation spécifié comme argument. Cette racine est le premier ancêtre stéréotypé par « view » ou « abstractView » et source d'une dépendance « viewExtension ».

```
viewRoot : ModelElement ;
-- récupération du parent direct dans P1
let P1=self.generalization.parent
if P1.clientDependency->select(isStereotyped("viewExtension"))->notEmpty
then viewRoot=P1
else viewRoot=P1.viewRoot
```

[5] La fonction *allParents* retourne un ensemble contenant les ancêtres d'une classe.

```
allParents : Set(ModelElement);
allParents = self.generalization.parent->union
(self.generalization.parent.allParents)
```

Annexe C : Sources J

Sources J du module VUMLProfile (extrait)

```
//-------------------------------------------------------------------------
// profile default#external#VUMLProfile
//-------------------------------------------------------------------------

void Package::checkModel ()
{

Package P = this;

StdOut.write(NL,"-----------------------------------------------",NL);
StdOut.write(NL,"Début de vérification de la cohérence du modèle",NL) ;

getAllClasses
   {
     StdOut.write("vérification de ",Name,NL) ;
     checkClass(P) ;

     // vérification des associations dont participe la classe courante
     // si elle est un <<view>> ou <<abstractView>>
     if (isStereotyped("view") or isStereotyped("abstractView"))
     {PartAssociationEnd
       {
          RelatedAssociation
          {
            checkAssociation();
          }
       }
     }
   }


StdOut.write("Fin de vérification de la cohérence du modèle",NL) ;

}   // method checkModel



void Class::checkClass (in Package CurrentPackage)
{

Class Base1;
Class Base2;
Class P1;
Class VRoot;
Class CurrentClass=this;
```

```
// Contraintes pour <<base>>

// [1] Un <<base>> a, au moins, une relation <<viewExtension>>, ou doit
être descendant direct d'un <<base>> ou d'un <<multiViewsClass>>

if (this.isStereotyped("base")) {
if ((CurrentPackage.getAllClasses.<select((isStereotyped("view") or
isStereotyped("abstractView"))
    && (DestinationUse.<select(isStereotyped("viewExtension") &&
UsedNameSpace==CurrentClass).size()!=0)).size()==0)
&& (ParentGeneralization.<select(SuperTypeClass.isStereotyped("base") or
SuperTypeClass.isStereotyped("multiViewsClass")).size()==0))

{
StdOut.write("Erreur : la classe ", Name," doit avoir des relations ") ;
StdOut.write("<<viewExtension>> ou être descendante ");
StdOut.write("d'une <<base>> ou un <<multiViewsClass>>",NL) ;
}
}


// [2] Un descendant direct de <<base>> est soit un <<base>> soit un
// <<multiVoewsClass>>

if ((!this.isStereotyped("base")) &&
(!this.isStereotyped("multiViewsClass")))
{if (ParentGeneralization.<select(
SuperTypeClass.isStereotyped("base")).size()>=1) {
StdOut.write("Erreur : la classe ", Name) ;
StdOut.write(" doit être stéréotpée soit par <<base>> soit par ") ;
StdOut.write("<<multiViewsClass>>",NL) ;
}
}


// Contraintes pour <<view>>

// [1] Un <<view>> a, au plus, un parent
if (this.isStereotyped("view")) {
if (ParentGeneralization.size()>1) {
StdOut.write("Le <<view>> ", Name) ;
StdOut.write(" doit avoir, au plus, un parent",NL) ;
}
}


// [2] Un <<view>> ne peut hériter que de <<view>> ou de <<abstractView>>
if (this.isStereotyped("view")) {
if (notVoid(ParentGeneralization.<select(
!SuperTypeClass.isStereotyped("view") &&
!SuperTypeClass.isStereotyped("abstractView"))) ) {
StdOut.write("Erreur : Le <<view>> ", Name) ;
StdOut.write("ne peut hériter que de <<view>> ou de <<abstractView>>",NL) ;
}
}
```

```
// [3] Un descendant direct d'un <<view>> est soit un <<view>> soit un
<<abstractView>>


if ((!this.isStereotyped("view")) && (!this.isStereotyped("abstractView")))
{if (ParentGeneralization.<select(
SuperTypeClass.isStereotyped("view")).size()>=1) {
StdOut.write("Erreur : la classe ", Name) ;
StdOut.write(" doit être stéréotpée soit par <<view>> soit par
<<abstractViewsClass>>",NL) ;
}
}


// [4] Un <<view>> doit être source d'une seule relation <<viewExtension>>,
ou descendant d'un et un seul
// <<view>> ou d'un et un seul <<abstractView>>

if (this.isStereotyped("view")) {
if ((DestinationUse.<select(isStereotyped("viewExtension")).size()!=1)
&&
(ParentGeneralization.<select(SuperTypeClass.isStereotyped("view") or
SuperTypeClass.isStereotyped("abstractView")).size()==0))
{
StdOut.write("Erreur : le <<view>> ",Name);
StdOut.write(" doit être source d'une seule relation <<viewExtension>>,");
StdOut.write(" ou descendant d'un et un seul <<view>> ou d'un et un seul
<<abstractView>>",NL) ;
}
}


// [5] Si un <<view>> est relié par une relation <<viewExtension>> à une
// base Base1, et hérite d'un <<view>> ou <<abstractView>> d'une base
// Base2, alors Base1 est un descendant de Base2


if (this.isStereotyped("view")) {
if ((DestinationUse.<select(isStereotyped("viewExtension")).size()==1)
&&
(ParentGeneralization.<select(SuperTypeClass.isStereotyped("view") or
SuperTypeClass.isStereotyped("abstractView")).size()!=0))

{ // Detérmination de la base de la vue
DestinationUse.<select(isStereotyped("viewExtension"))
{Base1=UsedNameSpace;
}

// Determination de la vue parent de la vue
ParentGeneralization.<select(SuperTypeClass.isStereotyped("view") or
SuperTypeClass.isStereotyped("abstractView"))
{P1=SuperTypeClass;}

// Detérmination de la vue racine de la vue parent

VRoot=P1.viewRoot();

// Detérmination de la base de la vue racine de la vue parent
VRoot.DestinationUse.<select(isStereotyped("viewExtension"))
{Base2=UsedNameSpace; }
Base1.inheritFrom(Base2);
```

```
if ((!Base1.inheritFrom(Base2)) or (Base1==Base2)) {
StdOut.write("Erreur générée par l'héritage entre la vue ",Name, " et la
vue ",P1.name, " : la <<base>> ", Base1.Name);
StdOut.write(" doit être descendante de la base de la vue ", P1.Name,NL) ;
}
}
}


// Contraintes pour <<abstractView>>

// [1] Un <<abstractView>> a, au plus, un parent
if (this.isStereotyped("abstractView")) {
if (ParentGeneralization.size()>1) {
StdOut.write("Le <<abstractView>> ", Name) ;
StdOut.write(" doit avoir, au plus, un parent",NL) ;
}
}


// [2] Un <<abstractView>> ne peut hériter que de <<view>> ou de
<<abstractView>>
if (this.isStereotyped("abstractView")) {
if (notVoid(ParentGeneralization.<select(
!SuperTypeClass.isStereotyped("view") &&
!SuperTypeClass.isStereotyped("abstractView"))) ) {
StdOut.write("Erreur : Le <<abstractView>> ", Name) ;
StdOut.write(" ne peut hériter que de <<view>> ou de <<abstractView>>",NL)
;
}
}


// [3] Un descendant direct d'un <<abstractView>> est soit un <<view>> soit
un <<abstractView>>

if ((!this.isStereotyped("view")) && (!this.isStereotyped("abstractView")))
{if (ParentGeneralization.<select(
SuperTypeClass.isStereotyped("abstractView")).size()>=1) {
StdOut.write("Erreur : la classe ", Name) ;
StdOut.write(" doit être stéréotpée soit par <<view>> soit par
<<abstractViewsClass>>",NL) ;
}
}


// [4] Un <<abstractView>> doit être source d'une seule relation
<<viewExtension>>, ou descendant d'un et un seul
// <<view>> ou d'un et un seul <<abstractView>>

if (this.isStereotyped("abstractView")) {
if ((DestinationUse.<select(isStereotyped("viewExtension")).size()!=1)
&&
(ParentGeneralization.<select(SuperTypeClass.isStereotyped("view") or
SuperTypeClass.isStereotyped("abstractView")).size()==0))
{
StdOut.write("Erreur : le <<abstractView>> ",Name);
StdOut.write(" doit être source d'une seule relation <<viewExtension>>,");
StdOut.write(" ou descendant d'un et un seul <<view>> ou d'un et un seul
<<abstractView>>",NL) ;  } }
```

```
// [5] Si un <<abstractView>> est relié par une relation <<viewExtension>>
// à une base Base1, et hérite d'un <<view>> ou <<abstractView>> d'une base
// Base2, alors Base1 est un descendant de Base2

if (this.isStereotyped("abstractView")) {
if ((DestinationUse.<select(isStereotyped("viewExtension")).size()==1)
&&
(ParentGeneralization.<select(SuperTypeClass.isStereotyped("view") or
SuperTypeClass.isStereotyped("abstractView")).size()!=0))

{ // Detérmination de la base de la vue abstraite
DestinationUse.<select(isStereotyped("viewExtension"))
{Base1=UsedNameSpace;
}

// Determination de la vue parent de la vue abstraite
ParentGeneralization.<select(SuperTypeClass.isStereotyped("view") or
SuperTypeClass.isStereotyped("abstractView"))
{P1=SuperTypeClass;}

// Detérmination de la vue racine de la vue parent

VRoot=P1.viewRoot();

// Detérmination de la base de la vue racine de la vue parent
VRoot.DestinationUse.<select(isStereotyped("viewExtension"))
{Base2=UsedNameSpace;
}

Base1.inheritFrom(Base2);
if ((!Base1.inheritFrom(Base2)) or (Base1==Base2)) {
StdOut.write("Erreur générée par l'héritage entre la vue ",Name, " et la
vue ",P1.name, " : la <<base>> ", Base1.Name);
StdOut.write(" doit être descendante de la base de la vue ", P1.Name,NL) ;
}
}
}


// Contraintes pour <<multiViewsClass>>

// [1] Un descendant direct de <<multiViewsClass>> est un
// <<multiViewsClass>> ou un <<base>>

if ((!this.isStereotyped("base")) &&
(!this.isStereotyped("multiViewsClass")))
{if (ParentGeneralization.<select(
SuperTypeClass.isStereotyped("multiViewsClass")).size()>=1) {
StdOut.write("Erreur : la classe ", Name) ;
StdOut.write(" doit être stéréotpée soit par <<base>> soit pas
<<multiViewsClass>>",NL) ;
} }
// Vérification des dépendances

checkDependancies();

}   // method checkClass
```

```
void Class::checkDependancies ()
{

Class S;
Class D;
Class DRoot;
Class SRoot;
Class DBase;
Class SBase;


// Vérification des <<viewExtension>>


//[1] La dépendance <<viewExtension>> a pour source un <<view>> ou
// <<abstractView>> et a pour destination un <<base>>.


DestinationUse {
if (isStereotyped("viewExtension")) {
// la dependance a pour source un <<view>> ou <<abstractView>>
// et une destination un <<base>>
if ((!UserNameSpace.isStereotyped("view")) &&
(!UserNameSpace.isStereotyped("abstractView"))) {
StdOut.write("Erreur, la source d'une dépendance <<viewExtension>>") ;
StdOut.write(" doit être un <<view>> ou <<abstractView>>",NL) ;
}
if (!UsedNameSpace.isStereotyped("base")) {
StdOut.write("Erreur : la destination d'une dépendance <<viewExtension>>")
;
StdOut.write(" doit être un <<base>>",NL) ;
}
}
}


// Vérification des <<viewDependency>>


//[1] La dépendance <<viewDependency>> a pour source un <<view>> ou
// <<abstractView>> et a pour destination un <<view>>

DestinationUse {
if (isStereotyped("viewDependency")) {
// la dependance a pour source un <<view>> ou <<abstractView>>
// et comme destination un <<view>>
if ((!UserNameSpace.isStereotyped("view")) &&
(!UserNameSpace.isStereotyped("abstractView"))) {
StdOut.write("Erreur, la source d'une dépendance <<viewDependency>>") ;
StdOut.write(" doit être un <<view>> ou <<abstractView>>",NL) ;
}
if ((!UsedNameSpace.isStereotyped("view"))) {
StdOut.write("Erreur : la destination d'une dépendance <<viewDependency>>")
;
StdOut.write(" doit être un <<view>>",NL) ;
}
}
}
```

```
// [2] Si la source d'une dépendance <<viewDependency>> a comme base B1 et
// si la destination de cette même dépendance a comme base B2, alors soit
// B2=B1 soit B1 est un descendant de B2.

DestinationUse {
if (isStereotyped("viewDependency")) {

// Detérmination de la vue source et la vue destination de la dépendance
S=UserNameSpace; D=UsedNameSpace;

//StdOut.write(NL, "S=", S.Name,"  D=", D.Name,NL) ;

// Detérmination de la vue racine des vues source et destination
SRoot=S.viewRoot();
DRoot=D.viewRoot();

//StdOut.write(NL, "SRoot=", SRoot.Name,"  DRoot=", DRoot.Name,NL) ;

SRoot.DestinationUse.<select(isStereotyped("viewExtension"))
{SBase=UsedNameSpace;
}

DRoot.DestinationUse.<select(isStereotyped("viewExtension"))
{DBase=UsedNameSpace;
}

// StdOut.write(NL, "SBase=", SBase.Name,"  DBase=", DBase.Name,NL) ;

if (!SBase.inheritFrom(DBase))  {
StdOut.write("Erreur générée par la relation <<vienDependency>> entre la
vue ",S.Name, " et la vue ",D.Name);
StdOut.write(" : les <<base>> des deux vues doivent être soit identiques
soit la base de la vue source est un descendant de celle");
StdOut.write(" de la vue destination",NL) ;
}
}
}

}   // method checkDependancies


boolean Class::inheritFrom (in Class Classe)
{

// Cette fonction vérifie si la classe en cours est une descendante de la
classe Classe

if (this==Classe) return true ;

this.ParentGeneralization.<select(SuperTypeClass.isStereotype("base") or
SuperTypeClass.isStereotype("multiViewsClass"))
{
 if (this.SuperTypeClass.inheritFrom(Classe)) return true; }

return false;

}   // method inheritFrom
```

```
boolean ModelElement::isStereotyped (in String stereotypeName)
{

// Cette fonction vérifie si l'élément de modélisation en cours est
// stéréotypé par stereotypeName

Stereotype stereotype;
stereotype = ExtensionStereotype;
if (notVoid (stereotype) ) then
return := (stereotype.name=stereotypeName) ;
else
return false;
endif

}   // method isStereotyped


boolean ModelElement::isStereokinded (in String stereotypeName)
{

// Cette fonction vérifie si l'élément de modélisation en cours est
// stéréotypé par stereotypeName ou l'un de ses ancêtres est stéréotypé par
// ce stéréotype

Stereotype stereotype;
stereotype = ExtensionStereotype;
if (notVoid (stereotype) ) then
return := stereotype.isName(stereotypeName);
else
return false;
endif

}   // method isStereokinded


Class ModelElement::viewRoot ()
{

// Cette fonction détermine la racine de l'élément de modélisation spécifié
// comme argument. Cette racine est le premier ancêtre <<view>> ou
// <<abstractView>> ayant une dépendance <<viewExtension>>.


if (DestinationUse.<select(isStereotyped("viewExtension")).size()==1)
return this;

ParentGeneralization.<select((SuperTypeClass.isStereotyped("view")) or
(SuperTypeClass.isStereotyped("abstractView"))) {
return this.SuperTypeClass.viewRoot();
}

}   // method viewRoot


void Association::checkAssociation ()
{
String A1;
String A2;
Class C1;
```

```
if (ConnectionAssociationEnd().<select(OwnerClass.isStereotyped("view") ||
OwnerClass.isStereotyped("abstractView")).size()>=1)

{

// récupération des classes participantes à la relation
ConnectionAssociationEnd()
{
 A1 = OwnerClass.Name;
}

ConnectionAssociationEnd() {
 if (OwnerClass.Name !=A1) A2=OwnerClass.Name;
}


// Contraintes pour <<view>> et <<abstractView>> relatives à la relation
// d'association

if (ConnectionAssociationEnd().<select(Aggregation==KindIsAssociation).size()>1)

{
// [1] Toute association entre des « view » ou des « abstractView » est
interdite.
if (ConnectionAssociationEnd().<select(OwnerClass.isStereotyped("view") ||
OwnerClass.isStereotyped("abstractView")).size()>1)
{
StdOut.write("Erreur : l'association ",A1,"-",A2, " est interdite : pas") ;
StdOut.write(" d'association entre <<view>> ou <<abstractView>>",NL) ;
}

else   // un seul participant est <<view>> ou <<abstractView>>

{ // Vérification de la contrainte [2] :
  // Si un « view » ou « abstractView » est relié par une association avec
  // une classe C (qui est ni un « view »
  // ni un « abstractView ») alors cette association doit être uniquement
  // navigable dans le sens « view »
  // ou « abstractView » vers C.

  // Récupération de l'unique <<view>> ou <<abstractView>> participant à
  // l'association
  ConnectionAssociationEnd().<select(OwnerClass.isStereotyped("view") ||
  OwnerClass.isStereotyped("abstractView"))
   {
    C1=OwnerClass;
   }


// Vérification si l'association est navigable de C1 vers l'autre
// extrémité
   if (ConnectionAssociationEnd().<select((OwnerClass==C1)&&
      (IsNavigable())).size()==0)
   {
    StdOut.write("Erreur : l'association ",A1,"-",A2);
    StdOut.write(" doit être navigable à partir de ",C1.Name,NL) ;
   }
}
}
```

```
else  {

// Contraintes pour <<view>> et <<abstractView>> relatives à la relation
// d'agrégation

if (ConnectionAssociationEnd().<select(Aggregation==KindIsAggregation).size()!=0)

{  // [1] Un « view » ou un « abstractView » ne peuvent jamais jouer le
   // rôle d'agrégés dans des relations d'agrégation.

if (ConnectionAssociationEnd().<select((OwnerClass.isStereotyped("view") ||
OwnerClass.isStereotyped("abstractView")) &&
(Aggregation!=KindIsAggregation)).size()=1)
{
StdOut.write("Erreur : l'agrégation ",A1,"-",A2, " est interdite : Un");
StdOut.write(" view » ou un « abstractView » ne peuvent jamais jouer le");
StdOut.write(" rôle d'agrégés",NL) ;
}

else   // l'agrégat est soit <<view>> soit <<abstractView>>

 { // [2] Si un « view » ou « abstractView » est relié par une relation
   // d'agrégation avec une classe C
   // (qui est ni un « view » ni un « abstractView »), alors la classe C
   // doit être l'agrégé de cette relation,
   // et cette agrégation doit être uniquement navigable dans le sens
   // « view » ou « abstractView » vers C.

  // Récupération de l'unique <<view>> ou du <<abstractView>> participant à
  // l'agrégation
  ConnectionAssociationEnd().<select(OwnerClass.isStereotyped("view") ||
  OwnerClass.isStereotyped("abstractView"))
   {
    C1=OwnerClass;
   }

  // Vérification si l'agrégation est navigable de C1 vers l'autre
  // extrémité
   if (ConnectionAssociationEnd().<select((OwnerClass==C1)&&
   (IsNavigable())).size()==0)
   {
    StdOut.write("Erreur : l'agrégation ",A1,"-",A2);
    StdOut.write(" doit être navigable à partir de ",C1.Name,NL) ;
   }
 }
}

else


// Contraintes pour <<view>> et <<abstractView>> relatives à la relation de
// composition
if (ConnectionAssociationEnd().<select(Aggregation==KindIsComposition).size()!=0)
{
// [1] Un « view » ou « abstractView » ne peuvent jamais être des
// composants dans des relations de composition.
if (ConnectionAssociationEnd().<select((OwnerClass.isStereotyped("view") ||
OwnerClass.isStereotyped("abstractView")) &&
(Aggregation!=KindIsComposition)).size()=1)
```

```
{
StdOut.write("Erreur : la composition ",A1,"-",A2, " est interdite : Un");
StdOut.write("« view » ou un « abstractView » ne peuvent jamais jouer le");
StdOut.write(" rôle de composant",NL) ;
}

}
}
}
}   // method checkAssociation
```

Sources J du module Code_Java (extrait)

```
//-----------------------------------------------------------------------
// profile default#external#Code#Java
//-----------------------------------------------------------------------

String Operation::getCode ()
{

// recapturing the content of all
// the "JavaCode" notes
DescriptorNote.<select (ModelNoteType.Name == "JavaCode")
{
// generation of the content
return.strcat (idTxt ());
return.strcat (Tab, Tab, Content, NL);
return.strcat (idEnd ());
}
// If there is no "JavaCode" note
// on the Operation, then markers are inserted
// allowing the automatic creation this type of
// text after external edition of the generated file
if (return == "") {
return.strcat (marker ("Descriptor", "JavaCode"));
}

}   // method getCode



String Operation::generate ()
{

int i=0;

if (isTaggedValue ("nocode") == false) {
// only methods without parameters should be generated
//if ((IOParameter.card() == 0) && (! (notVoid (ReturnParameter))))
//{
// generation of the method
return.strcat (idBox ());
return.strcat (NL);
// adding the word "synchronized" if the tagged
// value is positioned
if (isTaggedValue ("synchronized")) {
return.strcat ("synchronized "); }
```

```
return.strcat (getVisibility()," ");

// génération de la signature de la fonction

if (notVoid (ReturnParameter)) return.strcat (ReturnParameter.getType()," 
", Name,"(");
else  return.strcat ("void ", Name, "(");

if (IOParameter.card() != 0)
{     if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat (getType (), " ", Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (getType (), " ", Name); i=1;}
      else  return.strcat (",",getType (), " ", Name);
      }
      }
}

return.strcat (")", NL);

return.strcat (idEnd ());


// adding of the opening bracket
return.strcat (idGen ());
return.strcat ("{", NL);
return.strcat (idEnd ());
// generation of the method implementation
return.strcat(getCode ());
// adding of the closing bracket
return.strcat (idGen ());
return.strcat ("}", NL);
return.strcat (idEnd ());
}

}   // method generate



boolean Operation::existe (in Operation[] listeOperations)
{
// Cette méthode permet de vérifier si la méthode courante est déjà générée
// dans la classe ViewExtension

Operation M=this;
Operation [] operations1;
Operation [] operations2;
Operation [] operations3;
Parameter P1;
Parameter P2;
int i;


// Construction de la liste des opération ayant le même nom que
// l'opération M
listeOperations {
```

```
if (Name==M.Name) operations1.addElement(this);
}

if (operations1.size()!=0)
{

// Construction de la liste des opération ayant le même nom et le même type
// de valeur de retour
operations1 {

if(notVoid(ReturnParameter)&&notVoid(M.ReturnParameter)&&(ReturnParameter.g
etType()==M.ReturnParameter.getType()))
            operations2.addElement(this);
else if (!notVoid(ReturnParameter)&&!notVoid(M.ReturnParameter))
operations2.addElement(this);
}

if (operations2.size()!=0)
{
// Comparaison des paramètres de l'opération M avec les opérations de
// l'ensemble operations2
operations2 {
if (!notVoid(IOParameter)&&!notVoid(M.IOParameter)) return true;
if ((!notVoid(IOParameter)&&notVoid(M.IOParameter)) or
(notVoid(IOParameter)&&!notVoid(M.IOParameter))) return false;
if notVoid(IOParameter)&&notVoid(M.IOParameter)&&
(IOParameter.card()==M.IOParameter.card()))
{
for(i=0; i<IOParameter.size();i=i+1)
  {
    getItemSet(IOParameter,i,P1);
    getItemSet(M.IOParameter,i,P2);
    if (P1.getType()!=P2.getType()) return false;
  }
return true;
}
}
}
else return false;

}

else return false;

}   // method existe



String Operation::generateViewExtensionOpertaions ()
{

int i=0;

if (isTaggedValue ("nocode") == false) {
return.strcat (idBox ());
return.strcat (NL);
// adding the word "synchronized" if the tagged
// value is positioned
if (isTaggedValue ("synchronized")) {
return.strcat ("synchronized ");
}
```

```
return.strcat ("public ");


// génération de la signature de la fonction

if (notVoid (ReturnParameter))
return.strcat (ReturnParameter.getType()," ", Name,"(");
else  return.strcat ("void ", Name, "(");

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat (getType (), " ", Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (getType (), " ", Name); i=1;}
      else  return.strcat (",",getType (), " ", Name);
      }
      }
}

return.strcat (")", NL);

return.strcat (idEnd ());

// adding of the opening bracket
return.strcat (idGen ());
return.strcat ("{", NL);
return.strcat (idEnd ());
// generation of the method implementation
return.strcat("LeverAccesInterdit(~"",Name,"()","~"","); return;);",NL);
// adding of the closing bracket
return.strcat (idGen ());
return.strcat ("}", NL);
return.strcat (idEnd ());

}

}   // method generateViewExtensionOpertaions


String Operation::getVisibility ()
{
// Cette fonction retourne la visibilité d' une opération


String V;

if (Visibility == Public) V="public";
if (Visibility == Private) V="private";
if (Visibility == Protected) V="protected";
return V;
// END OF MODIFIABLE ZONE@OBJID@16108@255328704:149@E@354

}   // method getVisibility
```

```
String Operation::generateCallSwitchingMechanism (
      in Class baseClass,
      in Class[] views)
{
int i=0;
Class [] listViewsRedefWithCall;
Class [] listViewsRedefWithoutCall;


if (isTaggedValue ("nocode") == false) {
// generation of the method
// thanks to a dialog box
return.strcat (idBox ());
return.strcat (NL);
// adding the word "synchronized" if the tagged
// value is positioned
if (isTaggedValue ("synchronized")) {
return.strcat ("synchronized ");
}

return.strcat (getVisibility()," ");

// génération de la signature de la fonction

if (notVoid (ReturnParameter))
return.strcat (ReturnParameter.getType()," ", Name,"(");
else  return.strcat ("void ", Name, "(");

if (IOParameter.card() != 0)
{  if (IOParameter.card() == 1)
   IOParameter   {
   return.strcat (getType (), " ", Name);
      }
   else
   { i=0;
     IOParameter   {
     if (i==0) { return.strcat (getType (), " ", Name); i=1;}
     else  return.strcat (",",getType (), " ", Name);
    }
   }
}

return.strcat (")", NL);

return.strcat (idEnd ());

// adding of the opening bracket
return.strcat (idGen ());
return.strcat ("{", NL);
return.strcat (idEnd ());

// generation of the method implementation

// génération du code permettant de rediriger l'appel d'une méthode de la
// base redéfinie dans une vie avec un appel à la méthode de la base dans
// la méthode redefinie

// listViewsRedefWithCall : liste des vues où la méthode est redéfinie avec
// un appel à la méthode de base
// listViewsRedefWithoutCall : liste des vues où la méthode est redéfinie
// mais sans appel à la méthode de base
```

```
if (isRedefinedInViews(baseClass,views,listViewsRedefWithCall,
listViewsRedefWithoutCall)) {


// génération de l'appel qui sera utilisé par les vues où la méthode est
// redéfinie mais sans appel à la méthode de base
if (listViewsRedefWithoutCall.size() != 0)
{
return.strcat (NL,"if (");
if (listViewsRedefWithoutCall.size() == 1)
listViewsRedefWithoutCall {
return.strcat ("_ActiveView.equals(~"",Name,"~")");
}
else
      { i=0;
      listViewsRedefWithoutCall {
      if (i==0) { return.strcat ("(_ActiveView.equals(~"",Name,"~"))");
             i=1;
            }
      else  return.strcat ("||(_ActiveView.equals(~"",Name,"~"))");
                                                                  }
      }

return.strcat (")", NL);

return.strcat (Tab,"{ try {",NL);
return.strcat(Tab,Tab,"getView_",baseClass.Name,"().",Name,"(");

// génération des paramètres de l'appel de redirection (mêmes paramètres
// que ceux de la fonction)

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat(Name);
            }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (Name); i=1;}
      else  return.strcat (",",Name);
            }
      }
}
return.strcat (");", NL);

// génération de la partie catch

return.strcat (Tab,Tab,"}", NL);
return.strcat (Tab,"catch (AccesInterditException e) {", NL);
return.strcat (Tab,Tab,"System.out.println(e);", NL);
return.strcat (Tab,Tab,"}",NL);
return.strcat ("}",NL);


// génération de la partie qui va être exécutée si une vue n'a pas
// redéfinie la méthode
```

```
if ((listViewsRedefWithoutCall.size()!=views.size()) &&
(listViewsRedefWithCall.size()==0)) {

return.strcat ("else super.",Name,"(");



// génération des paramètres de l'appel de redirection (même paramètres
// que ceux de la fonction)

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat (Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (Name); i=1;}
      else  return.strcat (",",Name);
            }
      }
}
return.strcat (");",NL);
}

}


// génération de l'appel qui sera utilisé par les vues où la méthode est
// redéfinie avec appel à la méthode de base

if (listViewsRedefWithCall.size() != 0)
{
if (listViewsRedefWithoutCall.size() != 0)  return.strcat ("else {",NL);
return.strcat ("if (_i==2) { _i=1;  super.",Name,"(");

// génération des paramètres de l'appel de redirection (même paramètres
// que ceux de la fonction)

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat (Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (Name); i=1;}
      else  return.strcat (",",Name);
      }
      }
}
return.strcat ("); }", NL,NL);

return.strcat ("else {",NL);

return.strcat (NL,"if (");
if (listViewsRedefWithCall.size() == 1)
```

```
listViewsRedefWithCall {
return.strcat ("_ActiveView.equals(~"",Name,"~")");
                                                }


else
      { i=0;
      listViewsRedefWithCall {
      if (i==0) { return.strcat ("(_ActiveView.equals(~"",Name,"~"))");
            i=1;
           }
      else  return.strcat ("||(_ActiveView.equals(~"",Name,"~"))");
                                                                  }
      }


return.strcat (")", NL);

return.strcat ("{ _i++;", NL);
return.strcat (Tab,"try {",NL);
return.strcat(Tab,Tab,"getView_",baseClass.Name,"().",Name,"(");

// génération des paramètres de l'appel de redirection (même paramètres
// que ceux de la fonction)

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat (Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (Name); i=1;}
      else  return.strcat (",",Name);
      }
      }
}
return.strcat (");", NL);

// génération de la partie catch

return.strcat (Tab,Tab,"}", NL);
return.strcat (Tab,"catch (AccesInterditException e) {", NL);
return.strcat (Tab,Tab,"System.out.println(e);", NL);
return.strcat (Tab,Tab,"}",NL);
return.strcat ("}",NL);


// génération de l'appel de redirection concernant une vue qui ne redéfinie
// pas la méthode de la base

return.strcat ("else super.",Name,"(");

// génération des paramètres de l'appel de redirection (même paramètres
// que ceux de la fonction)

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
```

```
      IOParameter   {
      return.strcat (Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (Name); i=1;}
      else  return.strcat (",",Name);
      }
      }
}
return.strcat (");",NL);
return.strcat ("}",NL);
if (listViewsRedefWithoutCall.size() != 0)   return.strcat ("}",NL);

} }

else {

// génération du code permettant de rediriger l'appel des méthodes propres
// aux vues

return.strcat (Tab,"try {",NL);
return.strcat(Tab,Tab,"getView_",baseClass.Name,"().",Name,"(");

// génération des paramètres de l'appel de redirection (mêmes paramètres
// que ceux de la fonction)

if (IOParameter.card() != 0)
{
      if (IOParameter.card() == 1)
      IOParameter   {
      return.strcat (Name);
      }
      else
      { i=0;
      IOParameter   {
      if (i==0) { return.strcat (Name); i=1;}
      else  return.strcat (",",Name);
      }
      }
}
return.strcat (");", NL);

// génération de la partie catch

return.strcat (Tab,Tab,"}", NL);
return.strcat (Tab,"catch (AccesInterditException e) {", NL);
return.strcat (Tab,Tab,"System.out.println(e);", NL);
return.strcat (Tab,Tab,"}",NL);

}

// adding of the closing bracket
return.strcat (idGen ());
return.strcat ("}", NL);
return.strcat (idEnd ());}

}   // method generateCallSwitchingMechanism
boolean Operation::isRedefinedInViews (
      in Class baseClass,
```

```
      in Class[] views,
      inout Class[] listViewsRedefWithCall,
      inout Class[] listViewsRedefWithoutCall)
{

// Cette méthode permet de vérifier si la méthode en paramètre est une
// redéfinition d'une méthode de la base.
// elle permet aussi de déterminer la liste des vues redefinissant la
// méthode sans appeler la méthode de la base (listViewsRedef).
// et permet aussi de déterminer la liste des vues redefinissant la méthode
// avec appelle de la méthode de la base (listViewsNonRedef)

Operation M=this;

if (!M.existe(baseClass.PartOperation) )  return false;

views {

if (M.existe(this.PartOperation.<select(isStereotyped("redefined"))))
listViewsRedefWithCall.addElement(this);

else if (M.existe(this.PartOperation))
          listViewsRedefWithoutCall.addElement(this);

}

if ( (listViewsRedefWithCall.size()!=0)||
(listViewsRedefWithoutCall.size()!=0)) return true;

return false;

}   // method isRedefinedInViews


void Object::moduleInstall ()
{

StdOut.write ("Installation of Java module", NL);

}   // method moduleInstall


void Object::moduleUninstall ()
{

StdOut.write ("Uninstalling the Java module ", Name, NL);

}   // method moduleUninstall


String Attribute::generate ()
{

// generation of the current attribute
return.strcat (idBox ());
return.strcat (getType (), " ", Name, ";", NL);
return.strcat (idEnd ());

}   // method generate
String Attribute::getType ()
{
```

```
// returns the type of a Java attribute according to
// the modeled type
TypeGeneralClass
{
if (Name == "integer")
return = "int";
else if (Name == "real")
return = "float";
else if (Name == "String")
return = "String";
else
return = Name;
}

}   // method getType


void Class::generate (in Package CurrentPackage)
{

String content;
MpGenProduct genProduct;
String fileName;
Class baseClass;
Class CurrentClass=this;
Class[] views;

// displaying a message in the console
if (isStereotyped("base")) StdOut.write ("Base_");
StdOut.write (Name, " : ");

// generation of a comment at the beginning of the class
content.strcat (idGen ());
content.strcat ("// --------------------", NL);
content.strcat ("// Class ");
if (isStereotyped("base")) content.strcat("Base_");
content.strcat (Name, NL);
content.strcat ("// --------------------", NL);
content.strcat (idEnd ());

// generation of the class
content.strcat (idBox ());


if (isStereotyped("base")) content.strcat ("class Base_",Name,NL);
else if (isStereotyped("abstractView"))
      content.strcat ("abstract class ",Name,NL);
     else content.strcat ("class ",Name,NL);

// Une vue doit hériter de la classe ViewExtension si elle est reliée par
une relation viewExtension à une base

if (DestinationUse.<select(isStereotyped("viewExtension")).size()==1)
DestinationUse.<select(isStereotyped("viewExtension"))
{
// détermination de la base de la vue
baseClass=UsedNameSpace;
content.strcat (" extends ViewExtension_",baseClass.Name,NL);
}

content.strcat (idEnd ());
```

```
// generation of the opening bracket
content.strcat (idGen ());
content.strcat ("{", NL);
content.strcat (idEnd ());

// generation of the class attributes

content.strcat (NL,"// Attributs", NL);

PartAttribute
{
content.strcat(generate());
}

// generation of the class methods

// S'il s'agit d'une vue, il faut générer un constructeur a un seul
// argument de type Base

if (isStereotyped("view")) {

content.strcat (NL,"// Constructeur", NL);
content.strcat ("public ",Name,"(Base_",baseClass.Name," o) {",NL);
content.strcat ("super(o);",NL);
content.strcat ("}",NL);

}

// génération des méthodes de la classe

PartOperation
{
content.strcat(generate());
}

// generation of the closing bracket
content.strcat (idGen ());
content.strcat ("}", NL);
content.strcat (idEnd ());

// displaying a message in the console
StdOut.write ("generate", NL);

// recapturing a class generation work product
genProduct = getAnyProduct();
if (notVoid (genProduct))
{// creation of the generated file path
fileName.strcat(genProduct.getAttributeVal("path"),"/");
if (isStereotyped("base")) fileName.strcat("Base_");
fileName.strcat(Name,".",genProduct.getAttributeVal("suffix"));
// letting the file be managed by
// the class work product
genProduct.mngFile (fileName, content);

// S'il s'agit d'une base il faut aussi générer la classe centrale qui
// portera le même nom que la classe multivues
if (isStereotyped("base")) {

// récupération de la liste des vues

views = CurrentPackage.getAllClasses.<select
```

```
((isStereotyped("view") or isStereotyped("abstractView")) &&
(DestinationUse.<select(isStereotyped("viewExtension") &&
UsedNameSpace==CurrentClass).size()!=0)) ;

// génération de la classe ViewExtension  (classe contenant les méthodes
generateViewExtenstionClass(CurrentClass,views);

}

}
else
{StdOut.write ("The class ~"", Name, "~" has no Java work product", NL);}

}   // method generate


void Class::generatePrincipaleClass (
      in Class baseClass,
      in Class[] views,
      in Operation[] generateOperations)
{

String content;
MpGenProduct genProduct;
String fileName;
Class V;
Class [] listViews;

// displaying a message in the console
StdOut.write (Name, " : ");

// generation of a comment at the beginning of the class
content.strcat (idGen ());
content.strcat ("// --------------------", NL);
content.strcat ("// Class ", Name, NL);
content.strcat ("// --------------------", NL);
content.strcat (idEnd ());

// generation of the class
content.strcat (idBox ());
content.strcat ("class ", Name, " extends Base_",Name, NL);
content.strcat (idEnd ());

// generation of the opening bracket
content.strcat (idGen ());
content.strcat ("{", NL);
content.strcat (idEnd ());

// génération des méthodes et attributs de la classe centrale

// génération des variables dédiées à stocker les valeurs des arguments des
// constructeurs de vues

content.strcat(NL," // Liste des variables dédiées à stocker les valeurs
des arguments");
content.strcat(" des construceurs des vues", NL);

views {
V=this;
PartOperation.<select(isStereotyped("create"))
{
```

```
IOParameter {
content.strcat(getType(), " _", Name,"_",V.Name,";", NL);
}
}
}

// génération d'un constructeur sans arguments de la classe principale
content.strcat(NL,"// Constructeur",NL);
content.strcat("public ",Name,"() {",NL);
content.strcat("current_ViewExtension",Name,"= new
ViewExtension_",Name,"(this);",NL,NL);
content.strcat("// Remplissage de la liste des noms des vues",NL);
content.strcat("_ViewsNamesList_",baseClass.Name,".addElement(~"~");",NL);
content.strcat("_ViewsNamesList_",baseClass.Name,".addElement(~"Admin",Name
,"~");",NL);
views {
content.strcat("_ViewsNamesList_",baseClass.Name,".addElement(~"",Name,"~")
;",NL);
}
content.strcat("}",NL);
content.strcat(NL,"// Partie concernant la gestion des vues",NL);

// génération de l'attribut current_ViewExtension
content.strcat("Private ViewExtension_",Name,"
current_ViewExtension",Name,";",NL);
content.strcat(NL,"// Vecteur des vues",NL);
content.strcat("private Vector   _ViewsList_",Name," = new Vector();",NL);
content.strcat(NL,"// Vecteur des noms des vues",NL);
content.strcat("private Vector   _ViewsNamesList_",Name," = new
Vector();",NL);
content.strcat(NL,"// Vecteur des noms des vues désactivées par
l'administrateur",NL);
content.strcat("private Vector   _DesactivateViewsNamesList_",Name," = new
Vector();",NL);
content.strcat(NL,"// Vecteur des vues déjà créées",NL);
content.strcat("private Vector   _CreateViewsList_",Name," = new
Vector();",NL);
content.strcat(NL,"// Vue active",NL);
content.strcat("protected String   _ActiveView = new String(~"~");",NL);
content.strcat(NL,"// indices des vues dans la liste des vues
_ViewsList",Name,NL);
if (views.size()!=0) {
views {
content.strcat("int _index",Name,"=0;",NL);
}
content.strcat("int _indexAdmin",Name,"=0;",NL);
}

// génération de l'accesseur getView

content.strcat(NL,"// getView_",Name," : méthode d'accès à la vue
active",NL);
content.strcat("protected ViewExtension_",Name,"  getView_",Name,"()
{",NL);
content.strcat("return current_ViewExtension",Name,";",NL,"}",NL);


// génération de l'accesseur setView

content.strcat(NL,"// setView : méthode d’activation de vues",NL);
content.strcat("public boolean setView(String V) {",NL,NL);
```

```
content.strcat("//création de la vue si elle n'est pas encore créée",NL);
content.strcat("createView(V);",NL,NL);
content.strcat("//Traitement de l'activation de la vue",NL);
content.strcat("if (_DesactivateViewsNamesList_",Name,".contains(V))
{",NL);
content.strcat(Tab,Tab,"System.out.println(~"Désolé : cette vue est
désactivée !!~");",NL);
content.strcat(Tab,Tab,"return(false);",NL);
content.strcat(Tab,"}",NL);
content.strcat("else {",NL);
content.strcat(Tab,"if (V.equals(~"~") && (!V.equals(_ActiveView)))",NL);
content.strcat(Tab,Tab,"{
current_ViewExtension",Name,"=(ViewExtension_",Name,")_ViewsList_",Name,"(0
);",NL);
content.strcat(Tab,Tab," _ActiveView=~"~";",NL);
content.strcat(Tab,Tab,"}",NL);


if (views.size()!=0) {
views {
content.strcat(Tab,"if (V.equals(~"",Name,"~") &&
(!V.equals(_ActiveView)))",NL);
content.strcat(Tab,Tab,"{
current_ViewExtension",baseClass.Name,"=(ViewExtension_",baseClass.Name,")_
ViewsList_",baseClass.Name,".elementAt(_index",Name,");",NL);
content.strcat(Tab,Tab," _ActiveView=~"",Name,"~";",NL);
content.strcat(Tab,Tab,"}",NL);

}
}

content.strcat(Tab,"if (V.equals(~"Admin",Name,"~") &&
(!V.equals(_ActiveView)))",NL);
content.strcat(Tab,Tab,"{
current_ViewExtension",Name,"=(ViewExtension_",Name,")_ViewsList_",Name,".
elementAt(_indexAdmin",Name,");",NL);
content.strcat(Tab,Tab," _ActiveView=~"Admin",Name,"~";",NL);
content.strcat(Tab,Tab,"}",NL);
content.strcat(Tab,Tab,"return(true);",NL);
content.strcat(Tab,"}",NL);
content.strcat("}",NL);



// génération de la méthode createView()

content.strcat(NL,"// createView() : méthode de création de vues",NL);
content.strcat("private  void createView(String V) {",NL,NL);
content.strcat("if (V.equals(~"~"))",NL);
content.strcat(Tab,"{if (!_CreateViewsList_",Name,".contains(V))",NL);
content.strcat(Tab,Tab,"{ _ViewsList_",Name,".addElement(new
ViewExtension_",Name,"(this));",NL);
content.strcat(Tab,Tab,"_CreateViewsList_",Name,".addElement(~"~");",NL);
content.strcat(Tab,Tab,"}",NL);
content.strcat(Tab,"}",NL);


if (views.size()!=0) {
views {
content.strcat("else if (V.equals(~"",Name,"~"))",NL);
```

```
content.strcat(Tab,"{if
(!_CreateViewsList_",baseClass.Name,".contains(V))",NL);
content.strcat(Tab,Tab,"{ _ViewsList_",baseClass.Name,".addElement(new
",Name,"(this));",NL);
content.strcat(Tab,Tab,"_CreateViewsList_",baseClass.Name,".addElement(~"",
Name,"~");",NL);
content.strcat(Tab,Tab,"_index",Name,"=_CreateViewsList_",baseClass.Name,".
indexof(~"",Name,"~");",NL);
content.strcat(Tab,Tab,"}",NL);
content.strcat(Tab,"}",NL);
}
}

content.strcat("else if (V.equals(~"Admin",Name,"~"))",NL);
content.strcat(Tab,"{if (!_CreateViewsList_",Name,".contains(V))",NL);
content.strcat(Tab,Tab,"{ _ViewsList_",Name,".addElement(new
Admin",Name,"(this));",NL);
content.strcat(Tab,Tab,"_CreateViewsList_",Name,".addElement(~"Admin",Name,
"~");",NL);
content.strcat(Tab,Tab,"}",NL);
content.strcat(Tab,"}",NL);
content.strcat("else  {",NL);
content.strcat(Tab,"System.err.println(~"Erreur : la vue '~"+V+~"' est
inexistante !!~");",NL);
content.strcat(Tab,"System.exit(0);",NL);
content.strcat(Tab,"}",NL);

content.strcat("}",NL);

// génération des mécanismes d'aiguillages  (on utilise la liste des
méthodes générées dans la classe ViewExtension)

content.strcat(NL,NL,"// Mécanimses d'aiguillage des appels",NL);
generateOperations {
content.strcat(NL);
content.strcat(generateCallSwitchingMechanism(baseClass,views));

}

// generation of the closing bracket
content.strcat (idGen ());
content.strcat ("}", NL);
content.strcat (idEnd ());

// displaying a message in the console
StdOut.write ("generate", NL);


// recapturing a class generation work product
genProduct = getAnyProduct();
if (notVoid (genProduct))
{// creation of the generated file path
fileName.strcat(genProduct.getAttributeVal("path"),
"/",
Name,
".",
genProduct.getAttributeVal("suffix"));
// letting the file be managed by
// the class work product
genProduct.mngFile (fileName, content);
}
```

```
else
{StdOut.write ("The class ~"", Name, "~" has no Java
work product", NL);}

}   // method generatePrincipaleClass



void Class::generateViewExtenstionClass (
      in Class baseClass,
      in Class[] views)
{

String content;
MpGenProduct genProduct;
String fileName;
Operation [] generateOperations;  //l'ensemble des opérations générées pour
                                  // la classe ViewExtension
Class V;
Operation M1;

// displaying a message in the console
StdOut.write ("ViewExtension_",Name, " : ");

// generation of a comment at the beginning of the class
content.strcat (idGen ());
content.strcat ("// --------------------", NL);
content.strcat ("// Class ViewExtension_", Name, NL);
content.strcat ("// --------------------", NL);
content.strcat (idEnd ());

// generation of the class
content.strcat (idBox ());

content.strcat ("class ViewExtension_", Name, " extends
_ViewExtension",NL);

content.strcat (idEnd ());



// generation of the opening bracket

content.strcat (idGen ());
content.strcat ("{", NL);
content.strcat (idEnd ());


// génération d'une référence vers la base de la classe multivues et son
// accesseur
content.strcat ("protected Object _Base;",NL);
content.strcat ("private void setBase(Object o) {",NL);
content.strcat ("_Base=o;",NL);
content.strcat ("}",NL);


// génération du constructeur de la classe ViewExtension
content.strcat (NL,"public
View_Extension_",baseClass.Name,"(Base_",baseClass.Name," o) {",NL);
content.strcat ("setBase(o);",NL);
content.strcat ("}",NL);
```

```
views {
V=this;
// generation of the class methods

PartOperation.<select (!isStereotyped("create") &&
!isStereotyped("destroy"))
{ M1=this;
  if (!existe(generateOperations))
  {content.strcat(M1.generateViewExtensionOpertaions());
   generateOperations.addElement(this);
  }
}

}


// generation of the closing bracket
content.strcat (idGen ());
content.strcat ("}", NL);
content.strcat (idEnd ());

// displaying a message in the console
StdOut.write ("generate", NL);

// recapturing a class generation work product
genProduct = getAnyProduct();
if (notVoid (genProduct))
{// creation of the generated file path
fileName.strcat(genProduct.getAttributeVal("path"),
"/ViewExtension_",
Name,
".",
genProduct.getAttributeVal("suffix"));
// letting the file be managed by
// the class work product
genProduct.mngFile (fileName, content);

// génération de la classe centrale dans laquelle il y aura la génération
// des mécanismes d'aiguillage
generatePrincipaleClass(baseClass,views,generateOperations);

// génération de la vue administrateur
generateViewAdminClass(baseClass,views);

}

else
{StdOut.write ("The class ~"", Name, "~" has no Java work product", NL);}

}   // method generateViewExtenstionClass




void Class::generateViewAdminClass (
      in Class baseClass,
      in Class[] views)
{
String content;
```

```
MpGenProduct genProduct;
String fileName;
Parameter [] listParameters;
Class V;
int i;

// displaying a message in the console
StdOut.write ("Admin",baseClass.Name," : ");

// generation of a comment at the beginning of the class
content.strcat (idGen ());
content.strcat ("// --------------------", NL);
content.strcat ("// Class Admin", baseClass.Name, NL);
content.strcat ("// --------------------", NL);
content.strcat (idEnd ());

// generation of the class
content.strcat (idBox ());

content.strcat ("class Admin", baseClass.Name, " extends
ViewExtension",baseClass.Name, NL);

content.strcat (idEnd ());

// generation of the opening bracket
content.strcat (idGen ());
content.strcat ("{", NL);
content.strcat (idEnd ());


// génération des méthodes et attributs de la classe Admin

content.strcat (NL,"// Constructeur", NL);
content.strcat ("public Admin",baseClass.Name,"(Base_",baseClass.Name," o)
{",NL);
content.strcat ("super(o);",NL);
content.strcat ("}",NL);


// génération de la méthode (viewsInitiate) permettant d’initialiser les
// variables dédiées à stocker les valeurs des arguments des constructeurs
// des vues

content.strcat(NL,"// viewsInitiate()",NL);

// génération de l'entête de la fonction viewsInitiate()
content.strcat ("viewsInitiate(");

// récupération de tous les paramètres des constructeurs des différentes
// vues
views {
V=this;
 if (PartOperation.<select(isStereotyped("create")).size()!=0)
  {
   PartOperation.<select(isStereotyped("create")) {
   if (IOParameter.card() != 0)
     { IOParameter {
       listParameters.addElement(this);
       }

     }
```

```
   }

  }
}


// génération des paramètres de la méthode (viewsInitiate)

if (listParameters.size() != 0)
{
      if (listParameters.size() == 1)
      listParameters  {
      content.strcat (getType()," _", Name,"_",V.Name);
                                                        }
      else
      { i=0;
      listParameters {
      if (i==0) { content.strcat (getType()," _", Name,"_",V.Name); i=1;}
      else  content.strcat (",",getType()," _", Name,"_",V.Name);
                                                              }
      }

}

content.strcat (") {", NL);

// génération du corps de la fonction viewsInitiate()
views { V=this;
PartOperation.<select(isStereotyped("create")) {
if (IOParameter.card() != 0) {
IOParameter { content.strcat
("(",baseClass.Name,")_base._",Name,"_",V.Name," =_",Name,"_",V.Name,
";",NL);
             }
}
}
}

content.strcat ("(",baseClass.Name,")_base._viewsInitiate = True;",NL);
content.strcat ("}    // viewsInitiate",NL,NL);


// generation of the closing bracket
content.strcat (idGen ());
content.strcat ("}", NL);
content.strcat (idEnd ());

// displaying a message in the console
StdOut.write ("generate", NL);


// recapturing a class generation work product
genProduct = getAnyProduct();



if (notVoid (genProduct))
{// creation of the generated file path
fileName.strcat(genProduct.getAttributeVal("path"),
"/Admin",
baseClass.Name,
```

```
".",
genProduct.getAttributeVal("suffix"));
// letting the file be managed by
// the class work product
genProduct.mngFile (fileName, content);


}

else
{StdOut.write ("The class ~"", Name, "~" has no Java work product", NL);}

}   // method generateViewAdminClass



void Package::generate ()
{

Package P = this;

// displaying a message in the console
StdOut.write ("Generation of the package ", Name, NL);
// generation of the packages of the current package.
OwnedElementPackage.<generate();
// generation of the classes of the current package.

// génération de la classe _ViewExtension
generateViewExtenstion();

// génération de la classe AccesInterditException
generateAccesInterditException();

// Génération des autres classes
OwnedElementClass.<generate(P);

}   // method generate



void Package::generateViewExtenstion ()
{

// Cette fonction permet de générer la classe _ViewExtension Dont héritent
// toutes les classes ViewExtension générées

String content;
MpGenProduct genProduct;
String fileName;

// displaying a message in the console
StdOut.write ("_ViewExtension : ");

// generation of a comment at the beginning of the class
content.strcat (idGen ());
content.strcat ("// --------------------", NL);
content.strcat ("// Class  _ViewExtension", NL);
content.strcat ("// --------------------", NL);
content.strcat (idEnd ());
```

```
// generation of the class
content.strcat (idBox ());

content.strcat ("class _ViewExtension", NL);

content.strcat (idEnd ());

// generation of the opening bracket
content.strcat (idGen ());
content.strcat ("{", NL);
content.strcat (idEnd ());

// génération de la méthode LeverAccesInterdit

content.strcat ("protected",NL);
content.strcat ("final void LeverAccesInterdit(String methode) {",NL);
content.strcat ("throw new AccesInterditException(methode);",NL);
content.strcat ("}",NL);

// generation of the closing bracket
content.strcat (idGen ());
content.strcat ("}", NL);
content.strcat (idEnd ());

// displaying a message in the console
StdOut.write ("generate", NL);

// recapturing a class generation work product
genProduct = getAnyProduct();
if (notVoid (genProduct))
{// creation of the generated file path
fileName.strcat(genProduct.getAttributeVal("path"),
"/_ViewExtension",
".",
genProduct.getAttributeVal("suffix"));
// letting the file be managed by
// the class work product
genProduct.mngFile (fileName, content);


}

else
{StdOut.write ("The class ~"", Name, "~" has no Java work product", NL);}


}   // method generateViewExtenstion




void Package::generateAccesInterditException ()
{

String content;
MpGenProduct genProduct;
String fileName;

// displaying a message in the console
StdOut.write ("AccesInterditException : ");
```

```
// generation of a comment at the beginning of the class
content.strcat (idGen ());
content.strcat ("// --------------------", NL);
content.strcat ("// Class AccesInterditException", NL);
content.strcat ("// --------------------", NL);
content.strcat (idEnd ());

// generation of the class
content.strcat (idBox ());

content.strcat ("class AccesInterditException extends RuntimeException
",NL);

content.strcat (idEnd ());

// generation of the opening bracket
content.strcat (idGen ());
content.strcat ("{", NL);
content.strcat (idEnd ());

// génération de la méthode AccesInterditException

content.strcat ("public AccesInterditException(String methode) {",NL);
content.strcat ("super(~"Acces interdit dans la méthode :
~"+methode);",NL);
content.strcat ("}",NL);


// generation of the closing bracket
content.strcat (idGen ());
content.strcat ("}", NL);
content.strcat (idEnd ());

// displaying a message in the console
StdOut.write ("generate", NL);

// recapturing a class generation work product
genProduct = getAnyProduct();
if (notVoid (genProduct))
{// creation of the generated file path
fileName.strcat(genProduct.getAttributeVal("path"),
"/AccesInterditException",
".",
genProduct.getAttributeVal("suffix"));
// letting the file be managed by
// the class work product
genProduct.mngFile (fileName, content);

}

else
{StdOut.write ("The class ~"", Name, "~" has no Java work product", NL);}

}   // method generateAccesInterditException


void JavaProduct::generate ()
{

// Java code generation
OriginModelElement.<generate();
```

```
// updating of all the visualizers
updateAllEditors();

}   // method generate


void JavaProduct::initProduct (in MpGenProduct Product)
{

String ProductName;
String Suffix;
String Path;
// start of a session
sessionBegin ("Propagate", true);
if (notVoid (Product)) {
// getting back the values of the father work product.
ProductName = Product.Name;
Path = Product.getAttributeVal ("path");
Suffix = Product.getAttributeVal ("suffix");
// initialization of the current work product
setName (ProductName);
setAttributeVal ("path", Path);
setAttributeVal ("suffix", Suffix);
}
// end of the "Propagate" session
sessionEnd ();

}   // method initProduct



void JavaProduct::update (in MpGenProduct Product)
{

String ProductName;
String Suffix;
String Path;
// start of a session
sessionBegin ("Propagate", true);
if (notVoid (Product)) {
// getting the values of the parent work product.
ProductName = Product.Name;
Path = Product.getAttributeVal ("path");
Suffix = Product.getAttributeVal ("suffix");
// initialization of the current work product
setName (ProductName);
setAttributeVal ("path", Path);
setAttributeVal ("suffix", Suffix);
}
// deletion of all the files managed by the work product
deleteAllFiles ();
// end of the "Propagate" session
sessionEnd ();


}   // method update
```

```
boolean JavaProduct::mustPropagate ()
{
// the propagation is carried out for any modeling element
// that is associated to the current work product
// A similar work product will therefore be built for all
// the packages && classes
return = true;

}   // method mustPropagate



boolean JavaProduct::isPresent (in MpGenProduct Product)
{
// prevents from having a work product of the same type
// on a modeling element that would be of
// package or class type
if (Product.ClassOf == ClassOf)
return = true;
else
return = false;

}   // method isPresent


void JavaProduct::visualize ()
{

String fileName;
// construction of the complete path
fileName.strcat (getAttributeVal ("path"),
"\",
OriginModelElement.Name,
".",
getAttributeVal ("suffix"));
// message display in the console
StdOut.write ("Visualization of the file ", fileName,NL);
// internal visualization of the generated file
intVisuFileName (fileName);


}   // method visualize


void JavaProduct::edit ()
{

String fileName;
// construction of the complete path
fileName.strcat (getAttributeVal("path"),
"\",
OriginModelElement.Name,
".",
getAttributeVal("suffix"));
// message display in the console
StdOut.write ("File edition", fileName, NL);
// external edit of the generated file
extEditFileName(fileName);

}   // method edit
```

```
String JavaProduct::getIdLineComment ()
{

return = "// ";


}   // method getIdLineComment


boolean ModelElement::isStereotyped (in String stereotypeName)
{

// Cette fonction vérifie si l'élément de modélisation en cours est
// stéréotypé par stereotypeName

Stereotype stereotype;
stereotype = ExtensionStereotype;
if (notVoid (stereotype) ) then
return := (stereotype.name=stereotypeName) ;
else
return false;
endif

}   // method isStereotyped



String Parameter::getType ()
{

// returns the type of a Java attribute according to
// the modeled type
TypeGeneralClass
{
if (Name == "integer")
return = "int";
else if (Name == "real")
return = "float";
else if (Name == "String")
return = "String";
else
return = Name;
}

}   // method getType
```

Annexe D : Cahier des charges du système d'enseignement à distance

Le problème consiste à modéliser/informatiser le fonctionnement d'un Système d'Enseignement à Distance (SED dans la suite). Le SED est décentralisé sur plusieurs sites, chaque site étant géré par un responsable. La politique du SED est menée par un directeur et un conseil pédagogique qui regroupe le directeur, les responsables de sites, les représentants des enseignants et des représentants d'étudiants. Pour simplifier, nous considérerons que le SED permet à des étudiants à distance de suivre des formations en temps-différé. *Le cas des formations en temps-réel suivis à distance ne sera pas traité.*

a- Acteurs

Les acteurs (utilisateurs finaux) du SED au formation de son exploitation sont :

- les étudiants, qui s'inscrivent à des formations, suivent des formation à distance, passent des examens, …
- les enseignants auteurs de formations, qui produisent des formations et les mettent à jour, produisent les compléments de formation et la liste des ouvrages conseillés pour approfondir une formation, …
- Les enseignants tuteurs (peuvent être des auteurs de formations), qui assurent le suivi des étudiants, répondent à leur interrogation, posent et corrigent des examens, …
- les responsables de site qui font la mise à jour de la liste des formation disponibles, gèrent les inscriptions d'étudiants, …
- le conseil pédagogique (doté d'un président) qui réunit les jurys, fait le bilan pédagogique, définit la stratégie du SED, traite les litiges et les cas particuliers, …
- le directeur du SED qui, entre autres, définit la stratégie globale et produit les bilans de fonctionnement.

Les acteurs du système qui participent au développement du système sont :

- les analystes/concepteurs
- les programmeurs
- les intégrateurs/testeurs
- les ergonomes
- les pédagogues
- les commerciaux

Les acteurs du système sous maintenance sont :

- les mainteniciens (informaticiens)
- les pédagogues

Le SED gère un ensemble de sites (centres de formation). Chaque site offre une liste de formations appelée "catalogue" accessible sur le web.

D'une façon générale, une formation est caractérisée par un numéro d'identification unique, un titre (une ligne de texte), un nombre de crédits (équivalence ECTS[7]), un enseignant (auteur) responsable,

[7] European Credit Transfer System

un résumé de moins de cinquante lignes de texte, des pré-requis. Ces derniers peuvent être de type général (décrits par une ligne de texte, de façon standard, par exemple "niveau Mathématiques Première S") ou d'autres formations du SED. Une formation est divisé en chapitres. A chaque chapitre sont associés un support électronique et une liste d'exercices. De plus, à chaque formation est associé un commentaire textuel proposant un certain nombre de compléments (pointeurs vers des documents/cours relatifs, éléments d'approfondissement de certains concepts, etc…), des conseils méthodologiques, etc.

Une formation du SED est un cours en temps-différé dont les étudiants s'inscrivent à distance. Il possède en plus des caractéristiques décrites ci-dessus une liste de dates possibles pour l'examen qui se passe via Internet, et un prix à payer lors de l'inscription.

Tous les documents associés aux formations sont disponibles sur le Web. Les différents formats sont txt (texte simple), pdf (Adobe), doc (MS Word), ppt (MS Powerpoint), ps (Postscript). D'autres formats spécifiques sont possibles pour certaines formations et seront spécifiés dans le commentaire de formation, avec une indication des outils nécessaires à la consultation des documents.

Un exercice est caractérisé par un niveau de difficulté, une liste d'exercices pré-requis et une solution type. L'étudiant ne soumet pas sa solution au SED. Il peut consulter la solution donnée par l'enseignant responsable de la formation.

Un examen est constitué d'un sujet et d'un corrigé. Le corrigé est disponible une semaine après l'examen. Les examens des années précédentes sont archivés dans des annales accessibles aux étudiants suivant la même formation.

Un étudiant s'inscrit librement à des formations pouvant appartenir à des catalogues de différents sites du SED. Il paie au moment de l'inscription à une formation, qui se fait généralement en début d'année. Pour chaque formation à laquelle il est inscrit, on lui attribue un tuteur auquel il pourra envoyer des questions par email. Si l'étudiant souhaite avoir les crédits correspondant à une formation, il doit tout d'abord s'inscrire à l'examen de la formation à une date qu'il choisit dans la liste des dates possibles. Dès lors qu'il s'est inscrit, le fait de ne pas passer l'examen le jour prévu équivaut à une note égale à zéro. Un examen se déroule en temps limité via Internet, en présence distante du tuteur de la formation qui peut répondre à des questions de compréhension du sujet. Un étudiant inscrit peut démissionner d'une formation. Si cette démission intervient avant la date d'examen choisie, la moitié du montant de l'inscription lui sera remboursée.

L'obtention de crédits correspondant à différentes formations permet d'obtenir des niveaux validés par le SED (Bac+1, Bac+3, Bac +5, etc.). C'est le jury qui décide de ces niveaux, au vu des résultats aux différents examens. La liste des formations nécessaires pour obtenir les différents niveaux est donnée dans un document général disponible sur le Web. Le jury se réunit une fois par an. Pour chaque niveau, il existe aussi des filières (Physique, Chimie, Informatique, Mathématiques, Gestion d'entreprise, etc.)

On s'intéresse dans la suite aux activités des acteurs de ce système. Ces activités ne sont bien sûr pas toutes automatisables. Celles qui sont automatisables ne sont pas toutes effectuées dans le cadre du SED. La liste donnée ci-dessous n'est donc pas forcément exhaustive.

b- Activités des acteurs

Utilisateurs finaux :

Activités d'un étudiant :

- consulter le catalogue d'un site
- s'inscrire à un cours (inscription et paiement en ligne, avant la dernière date d'examen possible)
- consulter un exercice (et la solution si elle est donnée)
- envoyer un message à un enseignant tuteur (qui lui a été affecté)
- s'inscrire à un examen (avant la dernière date d'examen possible)
- passer un examen (à la date demandée lors de l'inscription)
- consulter son résultat à un examen
- consulter l'état global des enseignements suivis (cours suivis, examens passés, résultats …)
- démissionner d'un cours (par lettre ou message, avant la date d'examen choisie).

Activités d'un enseignant auteur :

- consulter les résumés des cours disponibles dans les différents catalogues
- consulter le contenu de ses propres cours
- demander au responsable de site d'ajouter un nouveau cours dans le catalogue
- demander au responsable de site la suppression de l'un de ses cours
- mettre à jour directement le support d'un cours existant (sans passer par le responsable)
- consulter la liste des étudiants inscrits à l'un de ses cours
- consulter des résultats d'examen (de ses cours ou non)

Activités d'un enseignant tuteur :

- consulter la liste des cours pour lesquels il assure le tutorat
- consulter la liste des groupes d'étudiants à suivre
- communiquer à l'auteur d'un cours des observations concernant ce cours
- proposer un sujet d'examen
- fixer une date d'examen
- supprimer une date d'examen
- assurer une séance d'examen (par une présence à distance)
- répondre à un message d'étudiant
- entrer les résultats d'examen d'un cours (dont il est tuteur)
- consulter les résultats à un examen (d'un autre cours)

Activités d'un responsable de site :

- ajouter un cours au catalogue (sur demande d'un enseignant auteur, après validation)
- supprimer un cours du catalogue (sur demande d'un enseignant auteur)
- consulter la liste des inscriptions à un cours donné
- traiter une démission d'étudiant (avec remboursement éventuel)
- consulter les résultats à un examen donné
- afficher les résultats d'un étudiant donné
- afficher les résultats de tous les étudiants inscrits
- afficher le chiffre d'affaire (montant des inscriptions moins les démissions) d'un cours
- afficher le chiffre d'affaire global du site (avec des détails)

- afficher le chiffre d'affaire d'un autre site (sans détail)

Activités du Conseil Pédagogique (via son président) :
- afficher les résultats pédagogiques d'un site (en fin d'année)
- afficher les résultats pédagogiques du SED global
- tenir un jury (en fin d'année)
- définir des orientations pédagogiques
- traiter un litige concernant un étudiant

Activités du directeur du SED :
- analyser les orientations pédagogiques du Conseil
- afficher les résultats pédagogiques d'un site
- afficher les résultats pédagogiques du SED
- afficher le chiffre d'affaire d'un site
- afficher le chiffre d'affaire global du SED
- produire un bilan global de fonctionnement du SED
- consulter le bilan d'une année précédente (3 dernières années possibles)
- définir la stratégie globale du SED

Acteurs du développement :

Activités du pédagogue
- proposer une stratégie didactique pour les niveaux et les filières (cours, inscriptions, examens, exercices, ...)
- proposer une liste de cours à développer
- définir des règles de suivi des étudiants

Activités de l'analyste/concepteur :
- étudier l'existant (systèmes équivalents, composants réutilisables, etc.)
- établir une conception du système (en respectant les spécifications du pédagogue)
- éditer/modifier des diagrammes de la conception

Activités de l'ergonome
- participer à l'analyse/conception (partie IHM)
- consulter les "diagrammes" liés à l'IHM

Activités du programmeur
- produire/générer du code à partir de la conception
- faire/documenter les tests unitaires associés
- éditer/modifier son code source

Activités d'un intégrateur/testeur
- intégrer/tester le système
- vérifier que les différents contenus sont lisibles et cohérents
- produire la documentation (manuel utilisateur...)

Activités du commercial
- faire une étude de marché et suggérer de nouveaux cours ou des suppressions de cours
- participer aux spécifications de cours (propositions de tarifs notamment)
- faire des simulations d'exploitation

- élaborer une plaquette commerciale
- faire de la publicité

Acteurs de la maintenance :

Activités du maintenicien

- consulter les documents de conception du système
- consulter le code source
- modifier l'analyse/conception
- modifier le code source
- tester le système modifié
- documenter les modifications

Activités du pédagogue

- analyser les bilans (pédagogiques)
- modifier la stratégie didactique
- analyser les pré-requis, les liens entre les cours ou vers les compléments
- demander l'ajout ou la suppression d'un cours
- demander la modification d'un cours (objectifs, contenu, liens, exercices, examen)

Annexe E : Code Java généré pour un extrait du diagramme de classes VUML du SED

```
/ --------------------
// Class  _ViewExtension
// --------------------
class _ViewExtension
{
protected
final void LeverAccesInterdit(String methode) {
throw new AccesInterditException(methode);
}
}



// --------------------
// Class AccesInterditException
// --------------------
class AccesInterditException extends RuntimeException
{
public AccesInterditException(String methode) {
super("Acces interdit dans la méthode : "+methode);
}
}



// --------------------
// Class Base_Formation
// --------------------
class Base_Formation
{

// Attributs
int id;
string titre;
int niveau;
int nbCredits;

// Méthodes
public void afficher()
{
}
}



// --------------------
// Class Formation
// --------------------
class Formation extends Base_Formation
{

// Constructeur
public Formation() {
current_ViewExtensionFormation= new  ViewExtension_Formation(this);
```

```
// Remplissage de la liste des noms des vues
_ViewsNamesList_Formation.addElement("");
_ViewsNamesList_Formation.addElement("AdminFormation");
_ViewsNamesList_Formation.addElement("EtudiantFormation");
_ViewsNamesList_Formation.addElement("EnseignantFormation");
_ViewsNamesList_Formation.addElement("ResponsableSiteFormation");
}

// Partie concernant la gestion des vues
Private ViewExtension_Formation current_ViewExtensionFormation;

// Vecteur des vues
private Vector   _ViewsList_Formation = new Vector();

// Vecteur des noms des vues
private Vector   _ViewsNamesList_Formation = new Vector();

// Vecteur des noms des vues désactivées par l'administrateur
private Vector   _DesactivateViewsNamesList_Formation = new Vector();

// Vecteur des vues déjà créées
private Vector   _CreateViewsList_Formation = new Vector();

// Vue active
protected String   _ActiveView = new String("");

// indices des vues dans la liste des vues _ViewsListFormation
int _indexEtudiantFormation=0;
int _indexEnseignantFormation=0;
int _indexResponsableSiteFormation=0;
int _indexAdminFormation=0;

// getView_Formation : méthode d'accès à la vue active
protected ViewExtension_Formation  getView_Formation() {
return current_ViewExtensionFormation;
}

// setView : méthode d’activation de vues
public boolean setView(String V) {

// création de la vue si elle n'est pas encore créée
createView(V);

// Traitement de l'activation de la vue
if (_DesactivateViewsNamesList_Formation.contains(V)) {
                        System.out.println("Désolé : cette vue est désactivée !!");
                        return(false);
            }
else {
if (V.equals("") && (!V.equals(_ActiveView))) {
current_ViewExtensionFormation=(ViewExtension_Formation)_ViewsList_Formation(0);
_ActiveView="";
}
if (V.equals("EtudiantFormation") && (!V.equals(_ActiveView)))
{
current_ViewExtensionFormation=(ViewExtension_Formation)_ViewsList_Formation.elementAt(_indexEtudia
ntFormation);
 _ActiveView="EtudiantFormation";
}
if (V.equals("EnseignantFormation") && (!V.equals(_ActiveView)))
```

```
{
current_ViewExtensionFormation=(ViewExtension_Formation)_ViewsList_Formation.elementAt(_indexEnseig
nantFormation);
 _ActiveView="EnseignantFormation";
}
if (V.equals("ResponsableSiteFormation") && (!V.equals(_ActiveView)))
                    {
current_ViewExtensionFormation=(ViewExtension_Formation)_ViewsList_Formation.elementAt(_indexRespo
nsableSiteFormation);
 _ActiveView="ResponsableSiteFormation";
}
if (V.equals("AdminFormation") && (!V.equals(_ActiveView)))
                    {
current_ViewExtensionFormation=(ViewExtension_Formation)_ViewsList_Formation.elementAt(_indexAdmin
Formation);
                     _ActiveView="AdminFormation";
                    }
                    return(true);
          }
}

// createView() : méthode de création de vues
private  void createView(String V) {

if (V.equals(""))
          {if (!_CreateViewsList_Formation.contains(V))
                    { _ViewsList_Formation.addElement(new ViewExtension_Formation(this));
                    _CreateViewsList_Formation.addElement("");
                    }
          }
else if (V.equals("EtudiantFormation"))
          {if (!_CreateViewsList_Formation.contains(V))
                    { _ViewsList_Formation.addElement(new EtudiantFormation(this));
                    _CreateViewsList_Formation.addElement("EtudiantFormation");
                    _indexEtudiantFormation=_CreateViewsList_Formation.indexof("EtudiantFormation");
                    }
          }
else if (V.equals("EnseignantFormation"))
          {if (!_CreateViewsList_Formation.contains(V))
                    { _ViewsList_Formation.addElement(new EnseignantFormation(this));
                    _CreateViewsList_Formation.addElement("EnseignantFormation");
                    _indexEnseignantFormation=_CreateViewsList_Formation.indexof("EnseignantFormation");
                    }
          }
else if (V.equals("ResponsableSiteFormation"))
          {if (!_CreateViewsList_Formation.contains(V))
                    { _ViewsList_Formation.addElement(new ResponsableSiteFormation(this));
                    _CreateViewsList_Formation.addElement("ResponsableSiteFormation");

          _indexResponsableSiteFormation=_CreateViewsList_Formation.indexof("ResponsableSiteFormation");
                    }
          }


else if (V.equals("AdminFormation"))
          {if (!_CreateViewsList_Formation.contains(V))
                    { _ViewsList_Formation.addElement(new AdminFormation(this));
                    _CreateViewsList_Formation.addElement("AdminFormation");
                    }
          }
```

```
else  {
        System.err.println("Erreur : la vue '"+V+"' est inexistante !!");
        System.exit(0);
        }
}


// Mécanismes d'aiguillage des appels

public void afficher()
{
if ((_ActiveView.equals("EtudiantFormation"))||(_ActiveView.equals("ResponsableSiteFormation")))
        { try {
                getView_Formation().afficher();
                }
        catch (AccesInterditException e) {
                System.out.println(e);
                }
}
else {
if (_i==2) { _i=1;  super.afficher(); }

else {

if (_ActiveView.equals("EnseignantFormation"))
{ _i++;
        try {
                getView_Formation().afficher();
                }
        catch (AccesInterditException e) {
                System.out.println(e);
                }
}
else super.afficher();
}
}
}

public void poserQuestion()
{
        try {
                getView_Formation().poserQuestion();
                }
        catch (AccesInterditException e) {
                System.out.println(e);
                }
}

public void ajouterExercice()
{
        try {
                getView_Formation().ajouterExercice();
                }
        catch (AccesInterditException e) {
                System.out.println(e);
                }
}

public void ajouterCorrection(Exercice exo)
{
```

```
	try {
		getView_Formation().ajouterCorrection(exo);
		}
	catch (AccesInterditException e) {
		System.out.println(e);
		}
}

public void repondreQuestion()
{
	try {
		getView_Formation().repondreQuestion();
		}
	catch (AccesInterditException e) {
		System.out.println(e);
		}
}


public void inscrireEtudiant()
{
	try {
		getView_Formation().inscrireEtudiant();
		}
	catch (AccesInterditException e) {
		System.out.println(e);
		}
}


public void affecterEnseignant()
{
	try {
		getView_Formation().affecterEnseignant();
		}
	catch (AccesInterditException e) {
		System.out.println(e);
		}
}

}



// --------------------
// Class ViewExtension_Formation
// --------------------
class ViewExtension_Formation extends _ViewExtension
{
protected Object _Base;
private void setBase(Object o) {
_Base=o;
}

public View_Extension_Formation(Base_Formation o) {
setBase(o);
}

public void afficher()
{
```

```
LeverAccesInterdit("afficher()"); return;);
}

public void poserQuestion()
{
LeverAccesInterdit("poserQuestion()"); return;);
}
public void ajouterExercice()
{
LeverAccesInterdit("ajouterExercice()"); return;);
}

public void ajouterCorrection(Exercice exo)
{
LeverAccesInterdit("ajouterCorrection()"); return;);
}

public void repondreQuestion()
{
LeverAccesInterdit("repondreQuestion()"); return;);
}

public void inscrireEtudiant()
{
LeverAccesInterdit("inscrireEtudiant()"); return;);
}

public void affecterEnseignant()
{
LeverAccesInterdit("affecterEnseignant()"); return;);
}
}



// --------------------
// Class EtudiantFormation
// --------------------
class EtudiantFormation
 extends ViewExtension_Formation
{

// Attributs
float prix;

// Constructeur
public EtudiantFormation(Base_Formation o) {
super(o);
}

// Méthodes
public void afficher()
{
}

public void poserQuestion()
{
}
}
```

```
// ---------------------
// Class EnseignantFormation
// ---------------------
class EnseignantFormation
 extends ViewExtension_Formation
{

// Attributs

// Constructeur
public EnseignantFormation(Base_Formation o) {
super(o);
}

// Méthodes
public void ajouterExercice()
{
}

public void ajouterCorrection(Exercice exo)
{
}

public void afficher()
{
}

public void repondreQuestion()
{
}
}
```

```
// ---------------------
// Class ResponsableSiteFormation
// ---------------------
class ResponsableSiteFormation
 extends ViewExtension_Formation
{

// Attributs
float prix;

// Constructeur
public ResponsableSiteFormation(Base_Formation o) {
super(o);
}


// Méthodes
public void inscrireEtudiant()
{
}

public void affecterEnseignant()
```

```
{
}

public void afficher()
{
}
}



// --------------------
// Class AdminFormation
// --------------------
class AdminFormation extends ViewExtensionFormation
{

// Constructeur
public AdminFormation(Base_Formation o) {
super(o);
}

// viewsInitiate()
viewsInitiate() {
(Formation)_base._viewsInitiate = True;
}    // viewsInitiate

}



// --------------------
// Class SupportDeCours
// --------------------
class SupportDeCours
{
// Attributs

// Méthodes
}


// --------------------
// Class Enseignant
// --------------------
class Enseignant
{

// Attributs
int id;
string nom;
string profil;


// Méthodes
}
```

```
// ---------------------
// Class Etudiant
// ---------------------
class Etudiant
{
// Attributs
int id;
string nom;
string adresse;

// Méthodes
}


// ---------------------
// Class Exercice
// ---------------------
class Exercice
{
// Attributs

// Méthodes
}



// ---------------------
// Class InfosInscription
// ---------------------
class InfosInscription
{
// Attributs
string dateDemande;

// Méthodes
}



// ---------------------
// Class Base_Question
// ---------------------
class Base_Question
{
// Attributs

// Méthodes
}



// ---------------------
// Class Question
// ---------------------
class Question extends Base_Question
{

// Constructeur
public Question() {
```

```
current_ViewExtensionQuestion= new  ViewExtension_Question(this);

//  Remplissage de la liste des noms des vues
_ViewsNamesList_Question.addElement("");
_ViewsNamesList_Question.addElement("AdminQuestion");
_ViewsNamesList_Question.addElement("EnseignantQuestion");
}

//  Partie concernant la gestion des vues
Private ViewExtension_Question current_ViewExtensionQuestion;

//  Vecteur des vues
private Vector   _ViewsList_Question = new Vector();

// Vecteur des noms des vues
private Vector   _ViewsNamesList_Question = new Vector();

//  Vecteur des noms des vues désactivées par l'administrateur
private Vector   _DesactivateViewsNamesList_Question = new Vector();

//  Vecteur des vues déjà créées
private Vector   _CreateViewsList_Question = new Vector();

//  Vue active
protected String   _ActiveView = new String("");

//  indices des vues dans la liste des vues _ViewsListQuestion
int _indexEnseignantQuestion=0;
int _indexAdminQuestion=0;

// getView_Question : méthode d'accès à la vue active
protected ViewExtension_Question  getView_Question() {
return current_ViewExtensionQuestion;
}

// setView : méthode d’activation de vues
public boolean setView(String V) {

// création de la vue si elle n'est pas encore créée
createView(V);

// Traitement de l'activation de la vue
if (_DesactivateViewsNamesList_Question.contains(V)) {
                        System.out.println("Désolé : cette vue est désactivée !!");
                        return(false);
          }
else {
          if (V.equals("") && (!V.equals(_ActiveView)))
                        { current_ViewExtensionQuestion=(ViewExtension_Question)_ViewsList_Question(0);
                         _ActiveView="";
                        }
          if (V.equals("EnseignantQuestion") && (!V.equals(_ActiveView)))
                        {
current_ViewExtensionQuestion=(ViewExtension_Question)_ViewsList_Question.elementAt(_indexEnseignant
Question);
                         _ActiveView="EnseignantQuestion";
                        }
          if (V.equals("AdminQuestion") && (!V.equals(_ActiveView)))
```

```
                    {
current_ViewExtensionQuestion=(ViewExtension_Question)_ViewsList_Question.elementAt(_indexAdminQue
stion);
                     _ActiveView="AdminQuestion";
                    }
                    return(true);
          }
}

// createView() : méthode de création de vues
private  void createView(String V) {

if (V.equals(""))
          {if (!_CreateViewsList_Question.contains(V))
                    { _ViewsList_Question.addElement(new ViewExtension_Question(this));
                    _CreateViewsList_Question.addElement("");
                    }
          }
else if (V.equals("EnseignantQuestion"))
          {if (!_CreateViewsList_Question.contains(V))
                    { _ViewsList_Question.addElement(new EnseignantQuestion(this));
                    _CreateViewsList_Question.addElement("EnseignantQuestion");
                    _indexEnseignantQuestion=_CreateViewsList_Question.indexof("EnseignantQuestion");
                    }
          }
else if (V.equals("AdminQuestion"))
          {if (!_CreateViewsList_Question.contains(V))
                    { _ViewsList_Question.addElement(new AdminQuestion(this));
                    _CreateViewsList_Question.addElement("AdminQuestion");
                    }
          }
else  {
          System.err.println("Erreur : la vue '"+V+"' est inexistante !!");
          System.exit(0);
          }
}


// Mécanismes d'aiguillage des appels


public void repondre()
{
          try {
                    getView_Question().repondre();
                    }
          catch (AccesInterditException e) {
                    System.out.println(e);
                    }
}
}


// --------------------
// Class EtudiantQuestion
// --------------------
class EtudiantQuestion
 extends ViewExtension_Question
{
```

```
// Attributs

public void creer()
{
}
}


// --------------------
// Class EnseignantQuestion
// --------------------
class EnseignantQuestion
 extends ViewExtension_Question
{
// Attributs

// Constructeur
public EnseignantQuestion(Base_Question o) {
super(o);
}

public void repondre()
{

}
}

// --------------------
// Class AdminQuestion
// --------------------
class AdminQuestion extends ViewExtensionQuestion
{

// Constructeur
public AdminQuestion(Base_Question o) {
super(o);
}

// viewsInitiate()
viewsInitiate() {
(Question)_base._viewsInitiate = True;
}    // viewsInitiate

}

// --------------------
// Class ViewExtension_Question
// --------------------
class ViewExtension_Question extends _ViewExtension
{
protected Object _Base;
private void setBase(Object o) {
_Base=o;
}

public View_Extension_Question(Base_Question o) {
setBase(o);
}

public void repondre()
```

```
{
LeverAccesInterdit("repondre()"); return;);
}
}

// ---------------------
// Class Reponse
// ---------------------
class Reponse
{

// Attributs

// Méthodes
}
```